

版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！

Broadview[®]
www.broadview.com.cn

[PACKT] open source*
PUBLISHING community experience distilled

Mastering Redis

深入理解Redis

让你的Redis技能产生质的飞跃，让开发酷炫应用从此轻而易举

[美] Jeremy Nelson 著
汪佳南 译



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Mastering Redis

深入理解Redis

[美] Jeremy Nelson 著

汪佳南 译

电子工业出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

本书以由浅入深、由原理到应用场景的方式介绍了 Redis 这款 NoSQL 数据库产品。书中不仅细致地讲解了 Redis 中的数据结构及流行的使用模式,还针对 Redis 键的设计和管理,以及内存管理提出了建设性的方案。同时,作者深入 Redis 源码,将其内部构造通过源代码调试的方式进行呈现。

本书适合有一定 NoSQL 经验的开发者或者架构师阅读。读者可以从书中找到许多应用场景和解决方案,例如 Docker 部署、Redis 消息队列、基于 Redis 的 ETL 应用和基于 Redis 的机器学习等。

Copyright © 2016 Packt Publishing.

First published in the English language under the title 'Mastering Redis'.

本书简体中文版专有出版权由 Packt Publishing 授予电子工业出版社。未经许可,不得以任何方式复制或抄袭本书的任何部分。专有出版权受法律保护。

版权贸易合同登记号 图字:01-2016-7238

图书在版编目(CIP)数据

深入理解 Redis / (美) 杰里米·尼尔森(Jeremy Nelson)著;汪佳南译. —北京:电子工业出版社,2017.4
书名原文: Mastering Redis

ISBN 978-7-121-31201-4

I. ①深… II. ①杰… ②汪… III. ①数据库—基本知识 IV. ①TP311.138

中国版本图书馆 CIP 数据核字(2017)第 065890 号

策划编辑:张春雨

责任编辑:郑柳洁

印 刷:北京中新伟业印刷有限公司

装 订:北京中新伟业印刷有限公司

出版发行:电子工业出版社

北京市海淀区万寿路 173 信箱 邮编:100036

开 本:787×980 1/16 印张:20 字数:416 千字

版 次:2017 年 4 月第 1 版

印 次:2017 年 4 月第 1 次印刷

定 价:89.00 元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010) 88254888, 88258888。

质量投诉请发邮件至 zlts@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式:010-51260888-819, faq@phei.com.cn。

译者序

对不少互联网企业来说，使用 LNMP 架构可以快速搭建起一套系统，产品可以迅速迭代并尽早投放市场。但随着访问量的上升，那些使用传统关系型数据库（例如 MySQL）的网站开始显现出性能方面的问题。此外，越来越挑剔的用户也要求网站不能仅专注于功能特性，同时也要追求极致的产品体验（即高性能和高可用）。

一方面，架构师会在数据库层面做以下一系列优化。

1. 配置主从

例如。为 MySQL 服务器配置主从。一台宕机，另一台可以顶上继续服务，以满足高可用的要求。

2. 读写分离

配置一主多从。因为一般系统 80% 的请求都是读取操作，将这些操作分发到从服务器上可以有效地提升网站响应速度。

3. 分库分表

随着数据量的增长，单库单表已经难以满足要求。一些诸如 TDDL、Cobar、MyCAT 等的 MySQL 分布式数据库中间件应运而生。

另一方面，架构师也会通过系统化的分析，以安插缓存层的方式缓解数据库的压力。对应用系统来说，缓存都应在设计之初就纳入考虑的范畴，成为系统不可或缺的一部分。我曾供职的一家公司就采用 Redis 作为缓存的实现方案。应用程序运行在两个同等的 Tomcat 容器里。Tomcat 之前则配置了一台 Nginx 用于 Load Balance。由于用户 Session 存储在 Redis 中（通过 Spring 和 Shiro 的简单配置即可实现），应用的无状态性使得系统可以很方便地进行水平扩缩。另外，我们的业务要求用户每次浏览某个项目页面时进行访问计数（counting）。

为了避免频繁的数据库读写，我们将每个项目的浏览量的计算和读取交给了 Redis。同时，为了避免意外导致的数据丢失，我们开启了 Redis 的持久化功能，并启动定时任务，每隔一段时间就将项目浏览次数同步到数据库中。

至于为何选择 Redis 而没有选择 Memcached，当时考虑的主要原因是 Redis 支持更为丰富的数据模型。像上述页面访问量计数的需求，可以直接通过 Redis 的命令进行操作，而且利用 Redis 的单线程模型，应用程序无须编写同步代码。当然，缓存只是 Redis 的其中一个用途。一些典型的场景还包括排行榜、用户访问量统计、集合运算、消息队列等。国内外的一些大型互联网企业（例如京东、新浪、Pinterest 等）都对 Redis 有不同程度的部署和应用。希望读者能够在本书中找到自己想要的答案。

每次翻译都带给我不同的体验，并为我的工作和生活带来改变。在此，我要感谢电子工业出版社的编辑张春雨和负责审校的同事，是你们的细心指导保证了本书的翻译质量。同时，感谢我的父母和我的太太，在你们的陪伴和支持下，我得以专心工作。

由于时间仓促，文中难免有所疏漏，望不吝斧正。

汪佳南

关于作者

Jeremy Nelson 是科罗拉多斯普林市的一所四年制私立文理学院科罗拉多学院的一位元数据和系统图书管理员。除了每周 8 小时的图书馆研究技术支持工作，为大学生提供信息素养指导，并监督图书馆的系统 and 编目部门这三项工作之外，Nelson 正在积极研究和开发 Catalog Pull 平台中的各种组件和开源工具，供科罗拉多大学、科罗拉多州研究图书馆联盟和国会图书馆使用。他还是语义网络初创公司 KnowledgeLinks.io 的联合创始人和 CTO。

他之前在西部州科罗拉多大学和犹他大学有过图书馆工作经验。在成为图书管理员之前，他曾在各种软件公司和金融服务机构中担任程序员和项目经理。他的第一本书 *Becoming a Lean Library* 于 2015 年出版，将精益创业和精益制造理念应用于图书馆和图书馆的运营。Nelson 从诺克斯学院获得了本科学位，并从 University of Illinois Urbana-Champaign 获得了图书馆和信息科学的科学硕士学位。

关于审校者

Emilien Kenler 在从事了一些小型 Web 项目之后,在 2008 年高中时开始专注于游戏开发。直到 2011 年,他为不同的小组工作并专门从事系统管理。

2011 年,在研究计算机科学工程的同时,他创立了一家公司销售 Minecraft 服务器。他基于像 Node.js 和 RabbitMQ 这样的新技术,创建了一个轻量级 IaaS (<https://github.com/HostYourCreeper/>)。

此后,他在 TaDaweb 担任系统管理员,构建基础架构并创建管理部署和监控的工具。

2014 年,他在东京 Wizcorp 开启了新的历程。同年,他毕业于 University of Technology of Compiègne。

Emilien 为 Packt Publishing 编写了 *MariaDB Essentials*。他还负责了 *Learning Nagios 4*、*MariaDB High Performance*、*OpenVZ Essentials*、*Vagrant Virtual Development Environment Cookbook* 和 *Getting Started with MariaDB-Second Edition* 的审校。

Saurabh Minni 拥有计算机科学专业的工程学位。他有超过 10 年的工作经验,通晓各种编程语言,包括汇编语言、C、C++、Java、Delphi、JavaScript、Android、iOS、PHP、Python、ZMQ、Redis、Mongo、Kyoto Tycoon、Cocoa、Carbon、Apache Kafka、Apache Storm 和 ElasticSearch。总之,他是一位彻头彻尾的程序员,喜欢每天学习与技术相关的新事物。

目前,他在 Near 公司(这是一家神奇的初创公司,正在搭建位置智能平台)中担任技术架构师一职。除了处理几个项目之外,他还负责部署 Apache Kafka 集群。这有助于简化大数据处理系统中的数据消费。这些系统包括 Apache Storm、Hadoop,等等。

Saurabh 同时也是 *Apache Kafka Cookbook* 一书的作者。这是一本有关 Apache Kafka 的书,由 Packt Publishing 出版。他还负责 *Learning Apache Kafka* 一书的审校。该书由 Packt Publishing 出版。

你可以在 Twitter 上通过@the100rabh 联系他，也可以在 <https://github.com/the100rabh/> 上找到他。

正是因为我的父母 Suresh 和 Sarla，以及我太太 Puja 的不断支持，才成就了本书。感谢有你们时刻陪伴在我身边。

前言

本书旨在从两方面为读者构建 Redis 的基础知识。一方面，本书提供了 Redis 及其技术背后的深层含义及理论；另一方面，拓展了 Redis 日常实用技能。本书书名中的精通（*Mastering*）二字暗示了精通 Redis 是一个持续的过程，而非最终目的地。激动人心的是 Redis 持续开放地演进成为了时下强大的数据操作和存储技术。

Redis 背后的设计哲学

在整个项目的生命周期中，Salvatore Sanfilippo 对 Redis 的发展方向与功能发表了独到的观点和见解。在 2015 年 1 月的一篇有关 Redis 对比其他数据库的基准测试的博客中，Sanfilippo 声明“我不想说服开发者们采用 Redis。我们只是尽力提供一款合适的产品。如果人们能够使用这款产品完成工作，我们会感到非常开心。这就是我的营销理念。”Sanfilippo 和他的 Redis 核心开发团队遵循着成功的开源管理模型：“仁慈独裁者”（BDL）模型。该模型中只能有单独一个人作为最终独裁者，来裁决哪些能被提交到 Redis 代码库中。BDL 模型的成功已经被诸如 Linux 内核开发和 Python 编程语言等项目证明过了。作为主要开发者和维护者的 Sanfilippo 成功地将 BDL 模型复制到了 Redis 中。

如果独裁者抛弃项目，或者更糟的是因病或者死亡而导致无法工作时，BDL 模型的失效将是灾难性的。Redis 浮现出来的另一个重大问题是当潜在的贡献者提交 Pull Request 时，针对这些提交的行动会被延迟，或者更多时候是被忽略。说句公道话，那些必须经过检查、测试并合并到主代码库上的变更的数量非常巨大，需要激情和专门的看门人。作为 Linux 内核项目的初创者及当前的 BDL，Linus Torvalds 已经看到自己的角色发生了转变，更多是在合并那些由其他开发者贡献的代码，同时相比亲自编写代码，他做的更多的是为 Linux 提供愿景和领导力。Sanfilippo 在 Redis 主要的电子邮件通信上的一篇主题帖中确认了该问题，他给出了以下两大主要理由来继续 Redis 当前的 BDL 模型：

- 项目开发与未来方向的一致性视角
- 对任何新的或者出现的更改采取问责制

在 Sanfilippo 看来, Redis 作为键值数据存储, 其易于部署、内存占用少 (就 Redis 本身来说, 而不是指它的数据集!)、安全可靠等特性一直是 Redis 在开发者和组织中人气持续上升的关键因素。他的观点确实造成了紧张的关系, 特别是当提出 Redis 的新功能, 例如使哈希中具体子值 (sub-values) 过期, 或者为可选功能提供可加载模块时, 这些特性都被拒之门外。Sanfilippo 对于保持 Redis 的小巧并专注于使其成为内存数据库的渴望, 推动着他的决策和开发实践。

在 2011 年的博客中, 他用有关 Redis 和 Redis 开发过程的七条宣言阐明了他的观点, 简述如下。

1. **用于抽象数据类型的 DSL**。Redis 是一门领域特定语言 (DSL), 用于抽象数据结构的表达和使用。这些数据结构不仅包括了操作 (Redis 命令), 还包括了使用 Redis 相关命令存储和操作这些数据结构的内存效率和时间复杂度。

2. **内存存储是第一要务**。通过将所有数据存储存储在计算机内存中, 跨系统的 Redis 的性能更为一致, 用于实现这些数据结构的众多算法以更为可预测的方式运行, 同时诸如有序集合这样的更为复杂的数据类型在内存数据库中更容易实现。

3. **基础数据结构对应基础 API**。Redis 为基础数据结构实现了一套基础 API。这套 API 由 Redis 命令和对应的数据结构组成。它尝试清晰地反映 API 从计算机内存中读取和写入的数据结构。Redis 遵循这一设计理念, 通过更简单的数据结构操作来组合实现更为复杂的操作。

4. **代码如诗**。这是所有七条宣言之中最难以渗透的。Sanfilippo 将他的审美偏好融入到代码之中以契合 Redis 这一长篇巨著。他认为 Redis 的编码风格和方法能够帮助人们书写一段叙述。因此, 是否包含第三方代码部分取决于它能否很好地契合 Redis 的叙述和 Redis 的源代码。

5. **我们反对复杂**。避免代码复杂性。如果要在使用大量代码来实现小型功能与舍弃该功能之间做出选择的话, 那么 Redis 会选择后者, 以舍弃额外的复杂性和向代码库中添加复杂性所带来的开销。

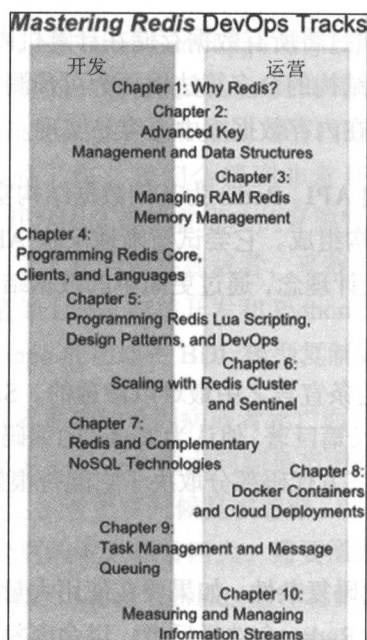
6. **两层 API**。一套是 API 的子集, 以分布式的方式运行; 另一套更大型、功能更为丰富, 用来支持多键操作。这种分离设计支持像 Redis 主从和 Redis 集群操作模式这样重要的功能。

7. 我们以优化为乐。通常对于开发者和技术运营者来说，这是一种情感诉求，也是一段非常聪明的声明。调优技术来解决疑难杂症所带来的刺激能够激起快乐的感觉及对 Redis 未来无限可能的兴奋。

本书涵盖内容

阅读本书时你会发现其贯穿着两大主题，以平行的方式展示了流行时尚的运营和过程的开发/运营二元论，即众所周知的 DevOps。为了有助于读者更有针对性地学习章节中包含的内容，每个章节的主题会被归类为软件开发或者系统运营。由于两者之间的边界越来越模糊，对每个趋势中主题的深刻理解能够增强你和你团队的能力，以便快速高效地为项目开发 and 部署 Redis 解决方案，或者将 Redis 作为技术基础设施需求的一部分。

在下图中，我们可以看到每个章节的水平位置展示了其偏向于软件开发还是系统运营。



DevOps 轨迹

第 1 章介绍了由 Redis 之父和主要维护者 Salvatore Sanfilippo 所阐明的 Redis 开发哲学。

第 2 章通过详细阐述和解释 Redis 数据结构和键管理为读者构建起 Redis 的基础知识，其中包括了如何为应用程序构造既有意义又有表达力的键模式这一重要主题。

第 3 章讲解了 Redis 提供的用来优化应用程序内存使用的各种选项，其中包括了 Redis 支持的基于最近较少使用（LRU）的各种缓存和 Redis 中的键驱逐（evict）策略。

第 4 章是有关应用程序编程的高级主题。本章从概述 Redis 核心的 C 语言实现开始，通过对精心挑选的 C 代码片段进行深入地讲解来加深你对 Redis 的理解。然后讲到了如何使用三种不同的 Redis 客户端，并展示了分别使用 Python、Node.js 和 Haskell 的编程示例。¹



第 5 章是有关应用程序编程的高级主题。本章从概述 Redis 服务器端 Lua 脚本及如何在 Redis 中更高效地使用 Lua 开始讲起。之后，拓展讲解了一些流行的 Redis 编程设计模式，列举了几个具体实例，描述了不同的个体和公司在运营操作中是如何使用这些模式的。本章最后从软件开发者的角度描述了 Redis 是如何应用于典型的 DevOps 场景中。

第 6 章探索了两个最近添加到 Redis 中的 Redis 集群和 Redis Sentinel。Redis Sentinel 是一种特殊的高可用模式，用来监控主从服务器的健康状况，并能够在任意 Redis 主从实例故障时进行切换。之前提到的 Redis 集群现在可用于产品环境中了。对于那些太大而不能存入单台机器的大型数据来说，Redis 集群能够通过运行多个实例并将键进行分片的方式来完成存储。虽然这些主题更关注运营方面，但是在采用 Redis 的工程解决方案时至少应该知道 Redis 群集的好处和局限性。

第 7 章一开始就承认了一件事，那就是对绝大多数组织来说，它们的信息技术栈包含了不同类型的数据和处理解决方案的异构混合。Redis 是一种用来扩展其他 NoSQL 数据存

¹ 译者注：本章中没有 Haskell 示例。

储选项的理想方式。同时，在本章中，我们将看到 Redis 是如何与 MongoDB、ElasticSearch 和 Fedora Digital Repository 一起使用的。那些需要使用多种解决方案来开发并支持复杂业务需求的开发者和系统管理员会对本章的内容感兴趣。

第 8 章介绍了如何通过 Docker 容器和镜像中使用 Redis 的方式来简化管理并提升基于 Redis 的解决方案的安全性和可靠性。Docker 是一种开源容器技术，正快速地被许多企业采用。在那之后我们将研究在最为流行的计算云供应商上使用 Redis 所面临的具体挑战，这些供应商包括最大、最为成熟的 Amazon Web Services、Google 的 Compute Engine 和 Microsoft Azure，并特别提到了其他云服务供应商，例如 Rackspace 和 Digital Ocean。本章最后会研究专门用于 Redis 的云服务。这些云服务专注于托管和管理 Redis 实例。

第 9 章一开始深入探索了 Redis 的发布/订阅命令。首先通过不同的示例展示了不同进程间、不同程序间、不同的 Redis 客户端、不同的操作系统，以及远程计算机上的发布者和消费者是如何进行通信的。之后，将拓展 Redis 的发布/订阅模式，并简要介绍将 Redis 用作企业计算生态中不同层之间的消息通信队列。本章最后会通过使用 Redis 和 Celery 作为任务管理和支持发布/订阅模式的消息通信队列这一详细示例总结提到的所有概念。

第 10 章在前面章节的基础上展示了 Redis 作为实时数据聚合器是如何将组织中来自各种技术系统的不同数据流聚合起来的。之后，将研究 Redis 安全模型和最新 Redis 版本新加入的安全特性。还有一款基于 Web 的运营仪表板将采用我们关于 Redis 客户端的知识，将输入到 Redis 中的数据进行可视化。接下来，将展示如何将机器学习算法（如朴素贝叶斯）应用到这些基于 Redis 的信息流，以提供更为丰富的概况，并加深你对组织或者部门内发生的操作的理解。

在附录的来源部分确认了章节中用到的摘录来源，并提供了链接以便读者进一步阅读。

获得精通 Redis 开放式徽章

Mozilla 基金会是赞助 Firefox 浏览器开发的开源组织，它启动了一项名为开放式徽章（Open Badges）的项目，允许组织创建并向个人颁发便携式和通用的徽章，以示其成就。



在本书网站上，你可以通过一系列在线测试获得精通 Redis 开放式徽章，并有机会向

你当前或者将来的雇主传达你对 Redis 日益增长的知识与技能。开放式徽章可以通过诸如 Facebook、Twitter 或者 LinkedIn 等流行的社交网站进行分享。

精通 Redis 开放式徽章对于购买了本书的读者来说是免费的。而对于那些没有购买的读者来说,仍然可以在本书网站上支付一些象征性的费用来获得精通 Redis 开放式徽章。你有机会与其他徽章获得者建立联系,从他们的经验中学习 Redis,同时可以分享自己的故事和知识,因此能够在阅读完本书之后的很长一段时间里鼓励你学习。我们希望本书能够马上帮助你理解 Redis,你能够通过获得开放式徽章记录这项专业成就。

需要为本书准备什么

Redis 设计用于在(诸如拥有现代 C++编译器)的 Linux 或者 Mac OS 等基于 POSIX 的环境下运行。当然,也有 Microsoft Windows 版本的 Redis,但是它并不受官方支持。更多信息请查看 <http://redis.io/download> 上有关 Windows 的这一节。本书中的示例使用了 Python 3.5,以及 Redis 的 Python 客户端(<https://github.com/andymccurdy/redis-py>)、Lua 和 Node.js,以及 Redis 的 Node.js 客户端(https://github.com/NodeRedis/node_redis)。

目标读者

如果你是一位 Web 开发者,对 MEAN 栈有基本的理解,有过 JavaScript 应用程序的开发经验,并且对 NoSQL 数据库也有基本的使用经验,那么本书就是为你而写的。

排版约定

你会发现本书采用了多种文本样式来区分不同种类的信息。下面是这些样式的一些示例和对应含义解释。文本中的代码、数据库表名称、文件夹名称、文件名、文件扩展名、路径名、虚拟 URL、用户输入和 Twitter handle 如下所示:“首先,在对键使用 EXPIRE 命令设置超时时,只有当删除或者替换该键时超时才会被清除。”

代码块如下所示:

```
def create_tea(datastore, name, time, size):
    # Increment and save global counter
    tea_counter = datastore.incr("global/teas")
    tea_key = "tea/{}".format(tea_counter)
    datastore.hmset(tea_key,
```

```
    {"name": name,  
     "brew-time": time,  
     "box-size": size})  
    return tea_key
```

任何命令行的输入或者输出如下所示：

127.0.0.1:6379> LATENCY HISTORY command

```
1) 1) (integer) 1433877379  
   2) (integer) 1000  
2) 1) (integer) 1433877394  
   2) (integer) 250
```



警告或者重要注意事项会出现在这个框中。



技巧和诀窍会出现在这里。

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），您即可享受以下服务。

- **提交勘误：**您对书中内容的修改意见可在【提交勘误】处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **与作者交流：**在页面下方【读者评论】处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/31201>

二维码：



目录

1 为何选择 Redis?	1
合适之选?	2
尝试使用 Redis	4
流行的使用模式	9
今非昔比	11
总结	13
2 高级键管理与数据结构	14
Redis 键	14
Redis 键模式	15
键分隔符和命名约定	17
手动创建 Redis 模式	19
解构 Redis 对象映射器	22
键过期	27
键的注意事项	27
大 O 符号	28
为自定义代码计算大 O 符号	30
回顾 Redis 数据结构的时间复杂度	32
字符串	32
哈希	33
列表	34
集合	35
有序集合	36
高级有序集合操作	39
位串和位操作	39

HyperLogLogs.....	41
总结	42
3 内存管理的建议与技巧	43
配置 Redis	43
主从复制	44
32 位 Redis	44
INFO memory 详解	46
键过期	48
LRU 键驱逐策略	53
创建内存高效的 Redis 数据结构	61
小巧的哈希、列表、集合和有序集合	61
把位、字节和 Redis 字符串用作随机访问数组	67
优化哈希，高效存储	68
硬件和网络延迟	71
操作系统建议	73
总结	74
4 Redis 编程第一部分：Redis 核心、客户端和编程语言	75
Redis 的内部结构	75
理解 redis.h 和 redis.c	82
Redis 序列化协议	92
Redis RDB 格式	95
使用 Redis 和 Python 创建协程	98
使用 Node.js 和 Redis 实现 Todo 列表应用	102
复制与公共访问	105
总结	105
5 Redis 编程第二部分：Lua 脚本、管理与 DevOps	106
在 Redis 中使用 Lua	106
使用 Redis 的 KEYS 和 ARGV	115

Redis 中的高级 Lua 脚本	119
MARC21 数据提取	119
纸质文具在线商店	121
让 JSON-LD、Lua 和 Redis 协同工作	124
Redis Lua 调试器	128
Redis 的编程与管理	131
主从复制	132
使用 MULTI 和 EXEC 实现事务	134
Redis 在 DevOps 中扮演的角色	137
总结	138
6 可伸缩性：Redis 集群和 Sentinel	140
数据分区的方法	140
范围分区	141
列表分区	143
哈希分区	146
复合分区	147
键哈希标签	148
使用 Twemproxy 实现 Redis 集群	149
使用关联数据片段服务器测试 Twemproxy	150
Redis 集群的背景	156
Redis 集群概览	157
使用 Redis 集群	158
Redis 集群实时重新配置及重新分片	163
故障转移	166
在 Redis 集群中替换或者升级节点	168
使用 Redis Sentinel 进行监控	169
为区域代码列表分区配置 Redis Sentinel	171
总结	174

7 Redis 与互补的 NoSQL 技术	175
NoSQL 技术的繁荣	175
Redis 作为 MongoDB 的分析补充	179
Redis 作为 Elasticsearch 的预处理组件	191
在 BIBCAT 中使用 Redis 和 Elasticsearch	191
ElasticSearch、Logstash 和 Redis	196
Redis 作为 Fedora Commons 的智能缓存补充	197
总结	203
8 Docker 容器与云端部署	204
Linux 容器	204
与 Redis 相关的 Docker 基础	209
Docker 镜像中的层	217
Docker 文件系统后端	218
Docker 和 Redis 的问题	225
使用 Docker Compose 打包应用程序	225
Redis 和 AWS	230
专门的云托管选项	231
Redis Labs	232
DigitalOcean Redis	232
总结	233
9 任务管理与消息队列	234
Redis 的发布/订阅模式概述	234
发布/订阅 RESP 回复	235
SUBSCRIBE 和 UNSUBSCRIBE RESP 数组	235
PSUBSCRIBE 和 UNSUBSCRIBE 数组	237
使用 redis-cli 进行发布/订阅	238
Redis 发布订阅实战	240
第一个工作站采用 Python 进行发布订阅	242

第二个工作站采用 Node.js 进行发布订阅	244
第三个工作站使用 Lua 客户端进行发布订阅	246
Redis 键空间通知	249
使用 Redis 和 Celery 进行任务管理	253
GIS 和 RestMQ	257
使用 RestMQ 进行任务管理	260
使用 Redis 技术进行消息通信	262
使用 Disque 进行消息通信	262
总结	264
 10 信息流的测量与管理	 265
基于 Redis 的 ETL 方案	265
将 JSON 转换成 RESP	271
管理 Redis 时的安全考虑	277
使用 Redis Web 仪表板进行运营监测	280
机器学习	281
朴素贝叶斯与工作分类	282
使用 Redis 实现线性规划	292
总结	296
 附录：来源	 298

为何选择 Redis?

为何选择 Redis? 或者说, 我们选择任何一项技术的缘由是什么? 每当新的技术或服务出现时, 我们总能听到那些勇敢的、愤世嫉俗的或是博学之人低声谈论这类问题。有时答案显而易见, 新的技术或服务所提供的特点和功能正好能满足我们迫切的需要, 或者能够解决棘手的问题。大多数情况下, 采用一门技术的理由并非显而易见, 有时深藏在晦涩难懂的营销行话之中。不同的需求左右着我们的选择, 你可能并不能通晓 Redis 的所有功能, 也不清楚其他公司是如何使用 Redis 的。Redis 不仅以执行速度快著称, 而且采用 Redis 构建的解决方案能够快速迭代, 这全靠 Redis 简单的配置、设置、运行和使用。

作为开源键值型 NoSQL 技术, Redis 越来越受欢迎离不开它的稳定性、灵活性及强大的功能。它在处理广泛的企业级数据操作时显得游刃有余。**REmote DIctionary Server (Redis)** 被一些公司广泛地采用, 这些公司包括初创公司及大型技术公司, 例如 Twitter 和 Uber, 再到个人和政府部门、学校和团体。本章将从 Redis 的一些流行的设计模式讲起, 然后为读者在是否采用 Redis 上提供实用的建议。

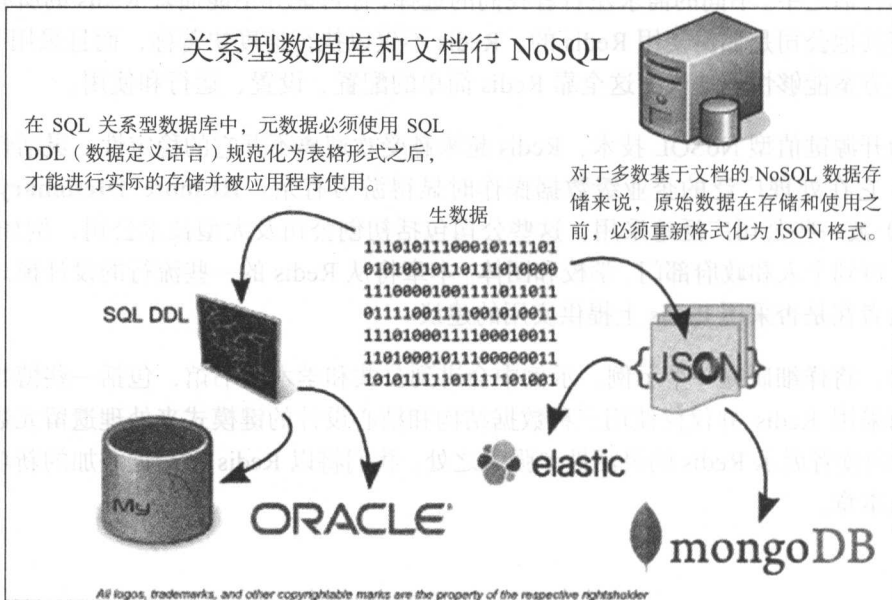
之后, 将详细研究一个示例。示例中会讲到公共和学术图书馆, 包括一些博物馆, 它们是如何采用 Redis 并仅仅使用三种数据结构和精心设计的键模式来处理遗留元数据格式的, 以此向读者展示 Redis 的灵活性和强大之处。我们将以 Redis 中最近添加的新功能和命令来结束本章。

合适之选？

在 Redis 邮件列表中一个相当常见的问题是询问对于广泛的使用场景来说，Redis 是否是合适之选。这些场景包括网站评论，缓存 MySQL 数据库的查询结果，或者说是否能够匹配提问者手头上的项目、产品、站点或者系统的特定需求。通常，Redis 是以快速读写数据见长的工具，它被大大小小的组织采用并获得了巨大的成功，其中的使用场景包含方方面面。Salvator Sanfilippo¹提供了有力的论据，那就是 Redis 无须替换现存的数据库，而在新功能开发或者解决一些难缠的问题方面，它对企业来说又是一种极好的补充。

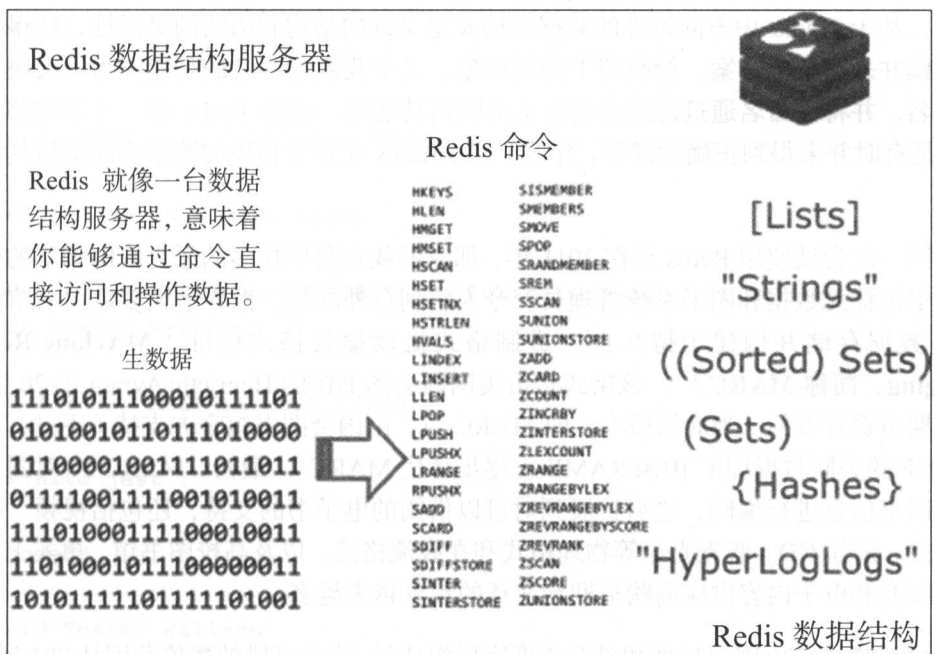
Redis 是单线程应用程序，占用较少的内存。它通过在数据中心和云供应商的多核处理器上运行多个实例达到持久性和可伸缩性。多台 Redis 组成的主从复制及正式发布的 Redis 集群，让运行多个 Redis 实例在内存和 CPU 需求方面的运营成本相对低廉。这就允许在兼顾伸缩性的同时，增强大型应用程序的持久性。

相较于典型的关系型数据模型，Redis 允许你以非常不同的方式概念化并解决富有挑战的数据分析及数据操控的问题。在基于 SQL 的关系型数据库中，开发者和数据库管理员通过将数据规范化为列、行和表，并通过外键关系建立关联的方式创建数据库模式。



¹ 译者注：Redis 之父。

其他像 MongoDB 和 Elasticsearch 这样的 NoSQL 数据存储技术需要在数据装载到实际存储之前转换为 JSON 文档数据格式。虽然 Redis 跳过了这种中间转换,但是在其他技术方面更进一步,它为特定的数据结构提供了一系列命令。这些数据结构包括字符串、列表、哈希表、集合及有序集合。这样的设计使得你可以通过算法和数据进行交互,以数据在 Redis 中的存储方式及可用的命令直接构造解决方案,同时能以更直接的方式对目标操作系统的内存和磁盘空间进行调优和监控。



思考诸如列表、哈希表和集合等基础计算数据结构是如何表达和管理的,有助于你以一种更基本的、更数学化的方式掌握数据及其结构的正反两面的特性。仔细思考中间结构化流程,例如将数据规范化为关系型数据库,或者为了使用 MongoDB 或 Elasticsearch 将数据转换为 JSON 文档。这样做虽然有价值,但是强制使用固定的数据结构,而 Redis 则没有这方面的要求。在构建解决方案时,你可能会发现你更需要那些非 Redis 技术支持的持久性和结构特性。即便在这种情况下,你对 Redis 中数据属性和结构的探索的经验,也有助于信息的处理和问题的解决。

当你拥有大量不经常使用的数据且无须立即存取时,Redis 可能不是最佳技术方案。基于 SQL 的关系型数据库或者文档存储型的 NoSQL 技术,例如 CouchDB 或者 MongoDB,相较 Redis 而言可能是更好的选择。但是,随着第三版 Redis 完全支持集群,在 Redis 中的

大型数据集可以作为分布式键值数据存储。越来越多的组织和个体从使用 Redis 集群中获取经验，期望会有更多的项目选择使用 Redis。

尝试使用 Redis

Redis 的数据类型丰富，很容易快速地对基于数据的算法进行实验。在我使用 Redis 的经验中，基于 Redis 中不同特性的数据结构及定义键的结构和语法的灵活性，使得我能够快速建模并采用解决方案。给我留下深刻印象，又令我激动不已的是能够为一大块可变的数据命名，并将该命名通过键的命名语义关联到其他键。这是 Redis 的一个非常伟大的功能，可是有时并未得到正确的评价。作为工具，Redis 在开发和理解数据方面是何其强大和有用。

我第一次尝试使用 Redis 是在 2011 年，那时的我在科罗拉多的派克峰山脚下的科罗拉多大学里担任元数据和图书系统管理员。令人感到意外的是，世界上大多数图书馆将它们的书目数据存储并构建于持久性二进制格式机读编目格式标准（**M**Achine-**R**eable **C**ataloging，简称 **MARC**）²。该格式是由美国国会图书馆的 Henriette Avram 于 20 世纪 60 年代后期负责开发的。当前的版本，即 MARC 21，由国会图书馆官方支持（不过，新一代基于 RDF 的关联数据词汇 BIBFRAME 正逐步替代 MARC）。最初，MARC 21 对图书馆书架上的图书信息进行编码，之后扩展到对可以借阅的电子书的支持，还包括视频、音乐和音频格式，诸如 CD、蓝光光盘等物理格式和在线流格式，以及高校图书馆。事实上，通过在线出版商和电子内容供应商购买期刊文章的预算越来越多。

MARC 格式是由固定长度和可变长度字段组成的，这些字段的数值范围从 001 到 999，相应的可以有字符数据或者含有数据的子字段。此外，每个字段可以包含最多两个指示符，用来修改字段的含义。MARC 域中最常用且最为重要的两个是 100 主要条目——个人名称字段和 245 题名说明字段。以下以 David Foster Wallace 的书 *Infinite Jest* 为例：

```
=100 1\ $aWallace, David Foster
=245 10$aInfinite jest :$ba novel$cDavid Foster Wallace
```

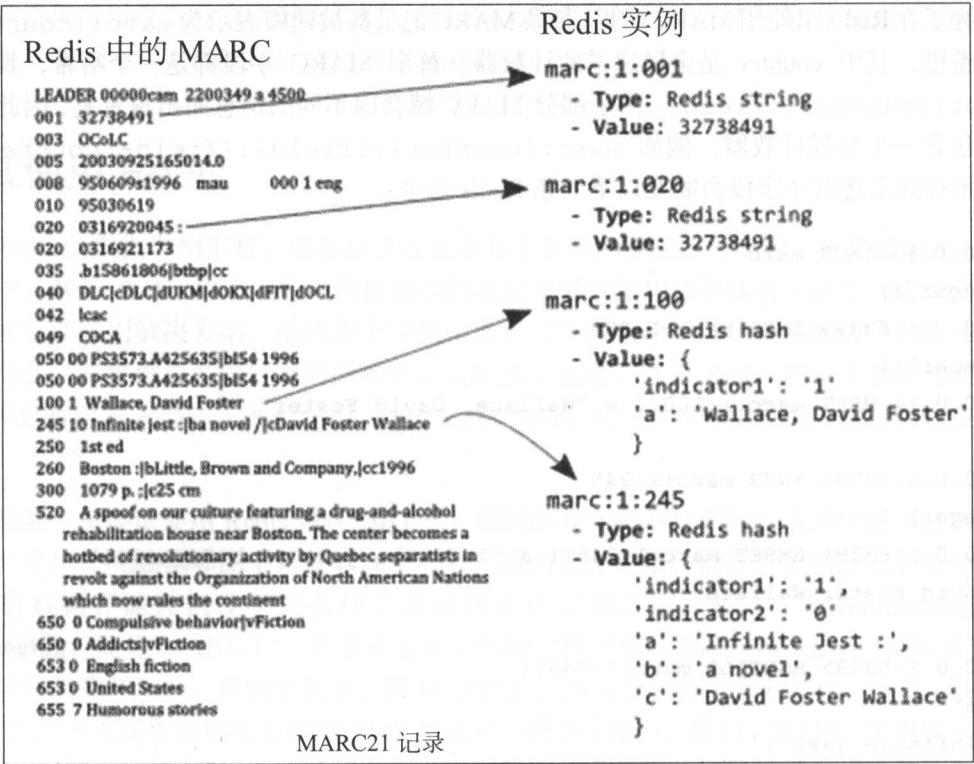
2 译者注：附百度文库 MARC 21 格式介绍

http://wenku.baidu.com/link?url=nsyDIRsolovEtb5s3Z9qm_Q9016j9oLOK2eW33lpuZS0GVEDrd7BeDIF98PLDeCIP5Oy_t1aOdgctxeLsPo863kuR2kkmkrIabLOthAUJKVG

为了在 Redis 里使用 MARC 数据, 每条 MARC 记录都被建模表示为 `marc:{counter}` 的哈希键, 其中 `counter` 是全局递增的计数器。每个 MARC 字段都是一个哈希, 其键为 `marc:{counter}:{field}`。由于部分 MARC 域会因不同的信息而出现重复, 因此哈希键会包含一个全局计数器, 例如 `marc:{counter}:{field}:{field-counter}`。简单地存储上述两个字段会需要以下 6 条 Redis 命令:

```
127.0.0.1> INCR marc
(integer 1)
127.0.0.1:6379> INCR marc:1:100
(integer 1)
127.0.0.1> HSET marc:1:100:1 a "Wallace, David Foster"
OK
127.0.0.1:6379> INCR marc:1:245
(integer) 1
127.0.0.1:6379> HMSET marc:1:245:1 a "Infinite jest :" b "a novel"
c "David Foster Wallace"
OK
127.0.0.1:6379> HGETALL marc:1:245:1
1) "a"
2) "Infinite jest :"
3) "b"
4) "a novel"
5) "c"
6) "David Foster Wallace"
```

Redis 中键的结构如下图所示。



Redis 中 MARC 数据存储可以仅通过单个 Redis 的哈希数据类型及一致的键语法结构完成。为了提升 Redis 中书目数据的实用性，并实现以书名和作者名字母数字排序的方式获取图书馆数据的列表记录(用图书馆的说法就是两个访问点)这样非常普通的用例场景，可以使用诸如列表或者有序集合等 Redis 中其他数据类型达成。

在 Redis 中，通过使用哈希和列表来表达 MARC 字段和子字段能提供更多信息。更进一步，我想知道 Redis 能否处理用于替代 MARC 的其他类型的图书和材料元数据模型。书目记录的功能性需求，或者简称为 FRBR，是一种可以替代 MARC 的文档，它基于实体-关系(ER)模型。FRBR ER 模型包含了根据抽象规则进行归类的属性组。最高抽象是 Work (作品)类，代表了用来唯一识别一件富有智慧的作品最为通用的属性，包括标题、作者和学科。

Expression (内容表达)类由诸如版本和与父 Work 有确定关系的译本组成。Manifestation (载体表现)和 Item (单件)是最后两个 FRBR 类，包含更多详细的数据，其中 Item 是一个具体的物理对象，是更为通用的 Manifestation 的具体实例。

仅有少数真实系统或者技术为图书馆数据实现了 FRBR 模型，而 Redis 提供使用真实数据测试此类模型的方法。采用将 MARC 数据映射到 FRBR 的 Work、Expression、Manifestation 和 Item 上的方法，那么 MARC 100 和 245 域就能在 Redis 中映射到 FRBR 的 Work。让我们通过 Redis 命令行工具 redis-cli 连接到 Redis 实例上来展示这些示例吧：

```
127.0.0.1:6379> HMSET frbr:work:1 title "Infinite Jest" "created by"
"David Foster Wallace"
OK
```

新的作品 frbr:work:1 可以通过下列 Redis 键和哈希从而关联到其余的类：

```
127.0.0.1:6379> HMSET frbr:expression:1 date 1996 "realization of"
frbr:work:1
OK
127.0.0.1:6379> HMSET frbr:manifestation:1 publisher "Little, Brown and
Company" "physical embodiment of" frbr:expression:1
OK
127.0.0.1:6379> HMSET frbr:item:1 'exemplar of' frbr:manifestation:1
identifier 33027005910579
OK
```

在之前 Expression 的例子中，具体日期是通过 **realization of** 属性关联回 frbr:work:1 的。同样，frbr:manifestation:1 哈希拥有两个字段：出版社和物质载体。物质载体字段的值是 frbr:expression:1 键，它将 Manifestation 关联回 Expression。最后的 frbr:item:1 哈希拥有一个条形码识别码属性和关联到 frbr:manifestation:1 哈希的键。

在对 MARC 和 FRBR 两者的实验中，Redis 哈希数据结构实体提供了基本表达。当具体的属性多于一个值时，例如当要表达作品的多个作者时，我们的策略就失效了。首次尝试解决多值属性引起的问题，是通过像之前那样为每个 MARC 字段创建一个计数器。举例来说，针对电子书或者其他拥有网络可解析 URL 的材料，MARC 856 字段（电子位置与存取）会为其存储 URL。如果想要为之前的 MARC 示例添加两个 URL，例如在 Google Books 里该书的链接地址及该书的 wiki 页面，那么对应的 Redis 命令如下所示：

```
127.0.0.1:6379> INCR global:marc:1:856
(integer) 1
127.0.0.1:6379> HMSET marc:1:856:1 ind1 4 ind2 1 u https://books.google.
com/books?id=Nhe2yvx6hP8C
OK
```

```
127.0.0.1:6379> HMSET marc:1:856:2 ind1 4 ind2 2 u http://infinitejest.
wallacewiki.com/
OK
```

针对 MARC 键的命名方法满足重复 MARC 字段的需求，但是如何能够支持一些极端情况，例如单一 MARC 域拥有多个、重复的子字段呢？要解决这个问题，首先想到的是可以存储以某种分隔符分隔的字符串，其中存储的每个值是 MARC 中特定字段的值。这就要求客户端进行额外的解析工作来获取所有不同的子字段，同时将失去将这些子字段直接存储至 Redis 所带来的额外优势。解决 MARC 多值子字段问题的第二种方法是进一步扩展 Redis 键语法，并为每个子字段使用列表或其他数据结构。让我们扩展 MARC 856 这个例子，如果想要添加第二个电子书的 URL，例如 Amazon Kindle 版本的 URL 链接，最终的 Redis 命令就会如下所示：

```
127.0.0.1:6379> LPUSH marc:1:856:1:u https://books.google.com/
books?id=Nhe2yvx6hP8C http://www.amazon.com/Infinite-Jest-David-Foster-
Wallace/
(integer) 2
127.0.0.1:6379> HSET marc:1:856:1 u marc:1:856:1:u
(integer) 0
```

将多个子字段存储到 Redis 列表这样的方法工作得很好，但是如果不希望 MARC 字段的子字段中没有重复的值该怎么办呢？这个问题可以通过使用 Redis 的集合数据类型轻而易举地解决。集合的定义就是只包含唯一的值。使用集合存储子字段的值看起来是个很棒解决方案，但如果需要在子域中存放有序值，集合就显得捉襟见肘了。

Redis MARC 子字段集合		
使用同样的 Redis 键，MARC 子字段可以存储为列表、集合或者是有序集合。每种选择各有利弊。		
	marc:1:856:u	
Redis 列表	Redis 集合	Redis 有序集合
<pre>[http://google.books.com/1, http://amazon.com/book/1, http://google.books.com/1]</pre>	<pre>{ http:amazon.com/book/1, http://google.books.com/1 }</pre>	<pre>{ (1, http://google.books.com/1) (2, http:amazon.com/book/1) }</pre>
优势： — 快速 — 保持有序	优势： — 所有的值都是唯一的 — 可用集合代数的概念进行运算	优势： — 所有的值都是唯一的 — 通过权重来保持有序
不足： — 允许重复值	不足： — 存储的值不是有序的	不足： — 三者当中最慢

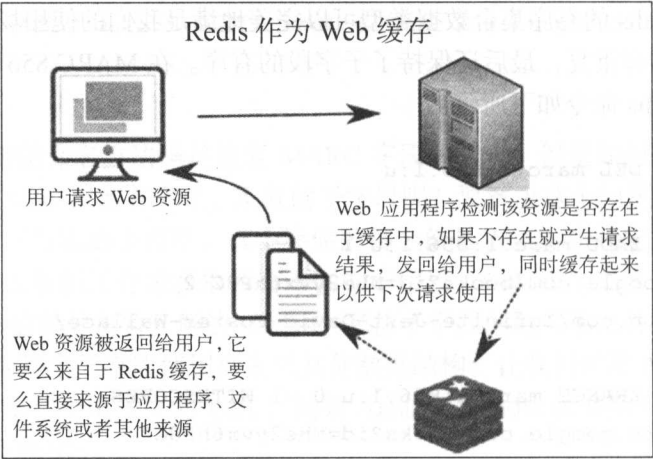
幸运的是, Redis 的有序集合数据类型可以完美地满足我们的使用场景。它确保集合当中子字段的唯一没有重复, 最后还保持了子字段的有序。在 MARC 856 字段中存储图书的 URL 所对应的 Redis 命令如下所示:

```
127.0.0.1:6379> DEL marc:1:856:1:u
(integer) 1
127.0.0.1:6379> ZADD marc:1:856:1:u 1
https://books.google.com/books?id=Nhe2yvx6hP8C 2
http://www.amazon.com/Infinite-Jest-David-Foster-Wallace/
(integer) 2
127.0.0.1:6379> ZRANGE marc:1:856:1:u 0 -1 WITHSCORES
1) "https://books.google.com/books?id=Nhe2yvx6hP8C"
2) "1"
3) "http://www.amazon.com/Infinite-Jest-David-Foster-Wallace/"
4) "2"
```

在该示例中, 我们研究了如何表示称为 MARC 的图书馆数据遗留格式, 以及 MARC 的字段和子字段如何以哈希的形式存储到 Redis 中。还有, 为了满足日益增长的需求, 针对子字段的存储方案不断改动, 从 Redis 的列表到集合, 最终使用了有序集合。希望这种迭代式的实验方法可以展示使用 Redis 的一个重要理由, 即能够快速测试数据存储的不同方法, 以及这些不同 Redis 数据类型 (哈希、列表、集合和有序集合) 的特征是如何被用来表示数据和存储, 以及访问该数据的需求。

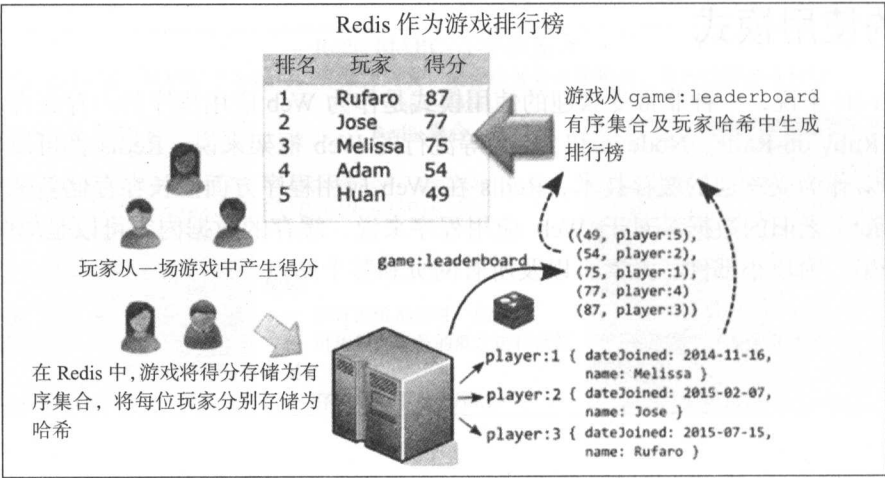
流行的使用模式

对 Redis 来说, 一种非常受欢迎的使用模式是作为 Web 应用程序的内存缓存。对诸如 Django、Ruby-on-Rails、Node.js 和 Flask 等流行的 Web 框架来说, Redis 都可以作为其缓存的选项。作为受欢迎的缓存技术, Redis 在 Web 应用程序方面擅长在存储新数据的同时驱逐 (evict) 老旧的数据。对于 Web 应用程序来说, 缓存的数据内容可以是单张 HTML 页面字符串、窗口小部件、元素, 以及所有网页和整个站点。



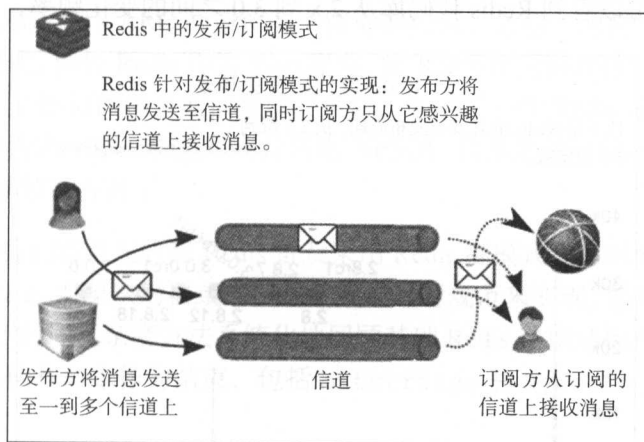
通过利用 Redis 为键设置过期时间的功能, 流行的 Redis 缓存策略之一——最近较少使用 (Less Recently Used, LRU) 策略变得非常健壮, 足以应对最大的网络站点。它将最受欢迎的内容保存在缓存中, 同时将陈旧的、较少使用的数据驱逐出数据存储。这种缓存的使用场景并不假定原始的 Web 元素或者页面是由 Redis 中的数据产生的。最常见的使用模式是由其他来源的数据动态生成 Web 内容, 而将 Redis 作为出色的 Web 缓存层。

第二种流行的使用模式是将 Redis 用作 Web 页面使用情况和玩家排行榜上的用户行为等定量数据的指标存储。通过在字符串上进行位操作, Redis 可以非常高效地将二进制信息存储于特定的字符串上。就拿网站使用来说, 有个从日期构造的键 `page-usage:2016-11-01`, 它对应一个字符串, 当该页面被用户访问时, 就将其中的一位设置成 1。



网站在 11 月 1 日的每日使用情况可以通过在 `page-usage:2016-11-01` 键上调用简单的 Redis 的 `BITCOUNT` 命令获取。在 2011 年的博客中，初创公司 Spool 中的几位员工详细讲解了他们是如何采用这种设计模式，使用位图及 Redis 的位操作来存储网站用户的活动的。

第三种流行的 Redis 使用模式是通过发布/订阅（简称 pub/sub）模型作为不同系统之间的通信层。在这种模型中，消息的发布方将消息发送至一到多个信道（channel）上，这些消息会被订阅或者监听信道上发来消息的其他系统处理。



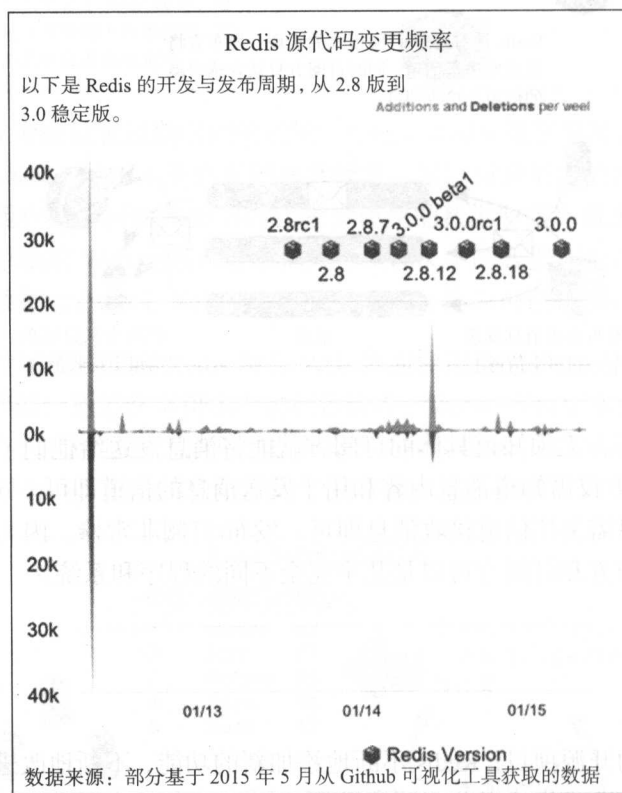
通常，消息发布方无须知道具体的订阅方就能将消息发送给他们（相对点对点消息通信模式来说），发布方仅需知道消息内容和用于发送消息的信道即可。同样，订阅方也无须知道每个发送方，只需关注信道接收消息即可。发布/订阅非常棒，因为它扩展起来非常容易，同时消息的发布方和订阅方可以是几乎完全不同的程序和系统。

今非昔比

作为一个活跃的开源项目，Redis 不断地添加新的功能，不断地改进并解决问题。这些问题可能正是在过去造成你或者公司同事觉得 Redis 不合适的原因。如果想要优化使用像 Redis 这样有价值且功能丰富的工具，就意味着你需要理解它的近期历史，并密切关注 Redis 最新稳定版本中包含的经开发、测试过的全新功能。Redis 遵循了常见的语义化的版本模式：`major.minor.patchlevel`。其中偶数的 `minor` 表示稳定版本，而奇数的 `minor` 表示不稳定的分支。

举例来说, Redis 2.8.9 发行版引入了两处更重要的改进。其中之一的 HyperLogLog 是一种高效的数据结构, 用来为唯一元素做数量估算。另一处则是为有序集合新增的 ZRANGEBYLEX、ZLEXCOUNT 和 ZREMRANGEBYLEX 命令。这两处改进将在第 2 章中进行深入详尽的讨论。随着 Redis 3.0 在 2015 年早期的发布, Redis 集群开始用于生产环境。它是 Redis 生态系统中最重要的一项功能, 将在第 7 章中进行详细讲解。

在下一个大版本 Redis 3.2 中, 将加入地理信息系统 (GIS) 命令及改进的有序集合, 支持 Redis 集群的全新 Lua 脚本, 以及全新的 Lua 调试器。将 Redis 代码库的变动频率进行可视化之后, 就可以看到 Redis 代码库从 2.x 到 3.0 之间的变化频率, 如下图所示。



当自问为何要选择 Redis 时, Redis 的充满活力、蓬勃发展的性质将是最好的答案。随着知识的持续积累及在精通 Redis 方面技能的不断提升, 曾经那些 Redis 的种种限制可能已不复存在。当你提升现有的技能并构建全新的、激动人心的机会去迎接未来时, 紧跟 Redis 的变化将成为第一要务。

总结

对于新项目或者你所要解决的数据问题来说, Redis 是否是一个正确的选择, 取决于数据的性质及想要在项目中尝试达成的目标。不同于关系型数据库或者文档型 NoSQL 数据库, Redis 并不要求在使用前将数据进行结构化。Redis 通过采用列表、哈希表、集合, 以及有序集合等多样化的数据结构, 提供一种直接的、更多算法的方式来操作数据。即便 Redis 未能成为你最终的选择, 通过将数据分解到数据结构的练习将有助于深化所要尝试解决的问题的语境。正如我们详细介绍的示例那样, 我们采用了哈希、列表、集合和有序集合等基础 Redis 结构表示成为 MARC 的遗留图书馆标准规范。之后, 简要回顾了使用 Redis 的三种流行的设计模式, 即将 Redis 作为 Web 缓存, 将 Redis 作为游戏排行榜的后端服务器, 以及将 Redis 用作发布/订阅消息通信系统。最后, 展示了一些 Redis 最近发生的变化。那些在过去一直采用传统 SQL 数据库或者其他 NoSQL 技术来解决的问题, 现在也可以将 Redis 作为主要数据解决方案了。

在第 2 章中, 我们将首先研究 Redis 键及采用 Redis 键模式来组织键的重要性。这些键模式可以通过由 Redis 对象映射器或是由手工文档进行规范来实现。然后, 将介绍大 O (Big O) 标记, 之后基于时间复杂度方法系统化地回顾基础 Redis 数据结构及命令。最后以介绍一些较新的数据结构和命令作为结束, 包括 bitstrings 和 HyperLogLog。

2

高级键管理与数据结构

在应用程序中将 Redis 用作数据存储需要先考虑以下两点：键和其对应的数据结构。制定一套良好的键模式、语法和命名约定是高效稳定的解决方案和技术混乱的分水岭。Redis 允许绝大多数的字符串经序列化后作为键，这种灵活性值得我们在设计基于 Redis 的项目时仔细斟酌与精心设计。对采用 Redis 构建的应用来说，为键选用合适的数据结构将对系统的可用性和功能性产生直接影响。本章包含以下内容：

- 设计并管理 Redis 的键模式和关联的数据结构。
- 使用 Redis 客户端对象映射器（Object Mapper）。使用不同的策略隐藏具体的键模式和数据结构。
- 使用 Javascript 的 Redis 对象映射器创建一个简单的应用，并分析对象映射器是如何使用 Redis 命令和数据结构的，以此作为 Redis 键模式的示例。
- 介绍大 O 标记，以及最坏情况的算法效率是如何应用在评估 Redis 命令的性能上的，还有性能是如何直接影响到 Redis 底层数据结构的。

对 Redis 官方文档中大 O 标记的理解，为估算基于 Redis 的应用程序的时间复杂度提供了一种方法，并有助于评估基于 Redis 的应用程序的性能。Redis 键值对应当作为解决方案的补充与加强，同时对应用程序的设计者、开发者和终端用户来讲，需要权衡是为了内存效率选择较短的键，还是为了解释清楚键的目的选择较长的键（足够详细）。

Redis 键

为了更有效地在应用程序中使用 Redis，我们需要理解 Redis 是如何存储键的，并了解

用于操作 Redis 实例中键空间的命令。运行 32 位还是 64 位版本的 Redis 将决定 Redis 键大小的实际限制。对 32 位版本来说,任何长于 32 位的键名需要更多的字节空间,因此增加了 Redis 的内存使用。使用 64 位版本的 Redis 允许更长的键长度,但是对于短小的键来说,也会分配完整的 64 位空间,从而导致额外的空间浪费。

Redis 的灵活性允许各种不同的键构造和存储方案。Redis 性能和可维护性的优劣取决于 Redis 数据库键的设计与构造。一种良好而又通用的实践是在设计 Redis 键时至少起草一份粗略概要,用于描述存入 Redis 中的信息及采用何种 Redis 结构中的初步想法。最后,你会想图形化表达 Redis 数据库中的数据结构是如何关联到存储在不同键上的其他信息的。这一过程通常归类在“Redis 键模式”规范下。不过,你的 Redis 键模式并不需要基于代码的方式来实现,只需要一份简单的文本来记录语法、键之间如何互相关联及每种键存储何种数据结构,这对于小型项目或用例场景来说就已经足够了。

Redis 键模式

虽然 Redis 官方教程 data types 推荐在命名键时使用一致的模式,但是 Redis 本身并没有模式检测或者验证的功能,不过我们仍然可以通过使用 EXISTS 和 TYPE 这些 Redis 命令实现一些基本的验证。如果应用需要明确特定类型的 Redis 键是否存在于实例中,可以通过使用 EXISTS 命令,随后使用 TYPE 命令确认该键是否是期望的 Redis 数据结构。除了这两个命令之外,验证 Redis 键语法和结构需要客户端代码(来实现)。

如果 Redis 应用会在不同的系统和组织中共享,那么为应用程序添加额外的验证逻辑层将十分有用。一份精确详尽的 Redis 键模式能极大地为你和应用开发者及运营人员在排除故障和调试问题方面带来帮助。另一种验证 Redis 键模式的方法是为 Redis 应用引入具体的单元测试,用来测试边界条件、模式键语法和结构,还有每个验证过的键所期望的数据结构。第三种验证 Redis 键模式的方法是使用 DTD 或者其他基于 XML 的键结构验证,或者使用新的键验证技术,例如 JSON Schema (<http://json-schema.org/>)。

验证 Redis 键的方案

虽然 Redis 会在对错误的数据结构进行操作时返回错误,但是 Redis 缺少正式的键模式验证机制。

选项一:通过应用程序的单元测试来测试 Redis 的键值结构

选项二:使用 Redis 对象映射器验证 Redis 键和正确的数据结构

选项三:使用像 JSON Schema 这样专门的测试或者验证工具



一份精心设计的键模式应当为现存的（基于 Redis 的）应用程序在添加新的键时提供指引。如果模式描述得足够清楚并且能够保持一致性的话，那么新 Redis 键的命名就不应当有任何神秘色彩。可以利用名词的单复数形式鉴别存储到 Redis 的实体内容与个数。举例来说，`book:1` 作为 Redis 哈希类型存储了单一书本的相关字段，而 Redis 键 `books:sci-fiction` 则存储了一套科幻类小说。有序集合可以用作图书销售排行，将 `books:sales-rank` 作为键名称，图书销售数量作为权重（有序集合的分值），并将图书键作为值。

对于一个简单的图书应用来说，基于文本的 Redis 模式示例如下所示：

名 称	Redis 数据类型	描 述	关 系
<code>book:{counter}</code>	哈希	存储了书名、作者、ISBN、格式、版权日期、页码数，以及价格等图书元数据	键存储在体裁集合和销售排行有序集合中
<code>books:{genre}</code>	集合	按照图书体裁进行分类的 Redis 键集合，例如通俗小说、推理小说、科幻小说和技术书籍	存储了所有按照单一体裁进行分类的图书键。与其他体裁集合一起，通过使用 <code>SINTERSTORE</code> 命令计算多体裁的图书，以及通过使用 <code>SDIFFSTORE</code> 计算单一体裁的图书
<code>books:sales-rank</code>	有序集合	存储每种图书的销售排行，以销售的名词作为有序集合的分值	存储所有 Redis 图书键的排名

即便是简单的、一次性的 Redis 项目，将键模式文档添加到项目的源代码仓库也是一种很好的实践。

键分隔符和命名约定

在上个示例中，我们用冒号“:”作为键的分隔符。对于复合 Redis 键来说，我们推荐使用冒号作为分隔符。不过，这只是一个约定而已，你可以在应用程序中使用任何分隔符。对 Web 应用来说，使用正斜杠“/”可能会更明智（当然，千万别直接把未经消毒预处理的用户 URL 请求直接传递给 Redis）。

另一种 Redis 键分隔符是英文字符句号“.”。大多数流行的编程语言，例如 C++、Object C、Python、Swift，以及 Ruby，都青睐这种面向对象的语法。只要保证分隔符的使用一致性，同时进行了适当的文档记录，就可以在 Redis 键模式中混合使用不同的分隔符。不论选择哪种分隔符，都应当向你和应用程序的最终用户保证清晰度和一致性。

高效的 Redis 键模式体现在建立命名约定以便将相关的键关联起来。应用程序和业务逻辑通过客户端代码应用在这些松散耦合的 Redis 键上。Redis 键模式以浅显易懂的方式将数据编织成一则故事，同时又能满足用户需求。就拿之前的图书键模式举例，扩展一下需求，让 Redis 数据库包含其他媒体类型，在 Redis 命令行工具中运行 KEYS 命令，并做一些格式化工作。我们可以从 Redis 键模式中观察到一种模式和隐含的关系：

```
all:sales-rank
global:book
book:1
book:2
book:3
books:genre:popular-fiction
books:genre:sci-fiction
books:format:ebook
books:format:paperback
books:sales-rank
global:film
film:1
film:2
films:genre:comedy
films:genre:drama
films:format:bluray
films:format:dvd
films:sales-rank
```

我们可以看到在此 Redis 应用程序中,图书和影片都提供了基本的前缀。那些用于支持的数据结构,通过基本前缀关联到单本图书或者单部影片,或者关联到包含额外实体哈希的集合上。

每件作品都是由 Redis 键作为前缀,同时加上一个全局计数的哈希。在此示例中,其他用于支持的数据结构,即图书和影片的系列与格式,都是 Redis 集合,其中存储了所有图书或者影片的键,并以特别的系列或者格式进行了分类。举例来说,Isaac Asimov 的 *Foundation* 的属性将存储在 book:2 哈希表中。同时,book:2 又是 books:genre:sci-fiction 和 books:format:paperback 集合的成员,也是 books:sales-rank 和 all:sales-rank 有序集合中的一项。同样地,Orson Well 的 *Citizen Kane* 存储在 film:1 哈希表中,同时也是 films:genre:drama 和 films:format:dvd 集合的成员,并且是 films:sales-rank 和 all:sales-rank 有序集合中的一项。

通常应用程序需要依据共同的特征获取集合的值。在图书示例中,我们尝试使用 KEYS 命令和 books:genre:* 模式获取所有的图书体裁。我们强烈建议不要在生产环境的应用中使用 Redis 的 KEYS 命令。这是因为 Redis 需要遍历数据库中每一个键。采用一致的命名约定及诸如集合、哈希或者有序集合这样的数据结构,应用程序理应无须使用 KEYS 命令获取数据。虽然 SCAN 命令可以用来获取 Redis 数据,但不应被视为 KEYS 命令的替代品。SCAN 命令抽取一个随机的键片段,然后将提供的模式和 MATCH 选项应用到此随机片段上。回顾之前的示例,下述 redis-cli 程序中运行的 SCAN 命令的用法仅对小型数据库有效:

```
127.0.0.1:6379> SCAN 0 MATCH books:genre*
1) "0"
2) 1) "books:genre:popular-fiction"
   2) "books:genre:mystery"
   3) "books:genre:teen"
   4) "books:genre:sci-fiction"
   5) "books:genre:fantasy"
   6) "books:genre:romance"
```

如果数据库再大一点,那么 SCAN 命令可能会无法匹配任何数据或者只返回所有匹配数据的一个子集。因此,将所有体裁的键存储在 books:genres 集合中才是明智之举,这样应用程序就可以像使用索引一样,使用 SMEMBERS 命令快速获取所有图书体裁的键:

```
127.0.0.1:6379> SMEMBERS books:genres
1) "books:format:ebook"
2) "books:genre:popular-fiction"
```

- 3) "books:format:paperback"
- 4) "books:genre:sci-fiction"
- 5) "books:sales-rank"

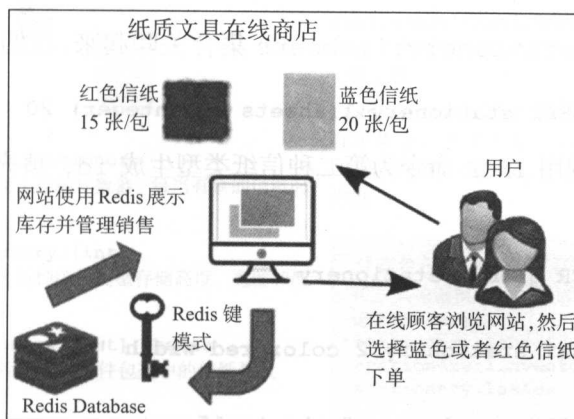
测试键之间的关系及它们是如何通过 Redis 键命名约定来互相关联的，这取决于众多因素，其中包括应用程序是否直接与 Redis 示例交互。添加单元测试来明确地检测 Redis 应用的键分隔符及命名约定，可以确保 Redis 数据库中存储的数据精确地表达了应用程序所依赖的假设和需求。以下是一个 Python 写的单元测试示例，用于测试图书示例中以冒号分隔符作为 Redis 键模式（完整示例请从 <http://mastering-redis.com> 下载）。

```
def test_delimiter(self):
    """Method tests for a colon in Redis keys in the datastore."""
    first_key = self.test_db.scan(0, "book*", 1)[1][0].decode()
    self.assertTrue(first_key.startswith("book:"))
```

程序中的第一行代码使用 Python 的 Redis 客户端以 `book:*` 模式运行 `scan` 函数，从初始游标 0 开始，同时把计数设置为 1，以便返回第一个实例，同时将字符串解码为 Unicode 并保存至 `first_key` 变量中。第二行断言变量 `first_key` 是以我们所期望的 Redis 模式中的前缀和分隔符开始的。接下来换一种场景，首先概述我们在 Redis 应用程序中想要捕获的主要数据类型，然后讨论如何使用业务需求和命名法则创建对应的 Redis 键和数据结构。

手动创建 Redis 模式

下图展示了一个售卖两种产品的在线商店的场景，我们将基于它构建 Redis 模式。



假设你经营着一家售卖纸制品的在线商店，售卖不同信纸产品。以这个简单的商业需求作为背景，接下来分几个独立的步骤介绍。

1. 网上的顾客浏览我们的网站准备购买信纸。

2. 有两种信纸可供选择：蓝色矩形包装的 20 张宣纸类信纸，以及红色矩形包装的 15 张宣纸类信纸。

3. 上述示例中的基本实体是信纸包裹，它有四种基本属性：颜色、高、宽，以及纸张的数目。（除非我们开始售卖非宣纸类信纸，否则我们将忽略材料属性。另一个将来会做的增强功能是为每个信纸包裹添加更友好的人名）。

4. 为了管理两种类型信纸的小型库存，当顾客从网站上购买纸品包装时，我们记录下时间和收到的数量，同时将库存减去售出的包裹数量。

手动创建 Redis 模式的第一步是建立全局信纸计数器，并附加在用于出售的信纸类型和品牌的信纸前缀之后。我们将颜色和尺寸属性存储为 `stationery:{id-counter}` 哈希中的字段，并将纸张数存储到另一个 `stationery:{id-counter}:sheets` 键所对应的字符串上。通过使用 `redis-cli` 程序连接本地运行的 Redis 后，就可以演示这些数据类型：

```
127.0.0.1:6379> INCR global:stationery1
```

返回的整数 1 将用作第一个信纸的 id：

```
127.0.0.1:6379> HMSET stationery:1 color blue width '30 cm' height  
'40 cm'OK
```

为了将纸张数和 `stationery:1:sheets` 集合关联起来，我们使用 `INCRBY` 命令：

```
127.0.0.1:6379> INCRBY stationery:1:sheets 20(integer) 20
```

现在，我们再次调用 `INCR` 命令为第二种信纸类型生成 id，填充哈希并增加 15 张信纸：

```
127.0.0.1:6379> INCR global:stationery  
(integer 2)  
127.0.0.1:6379> HMSET stationery:2 color red width '45 cm' height '45 cm'  
15  
127.0.0.1:6379> INCRBY stationery:2:sheets 15
```

(integer) 15

接下来, 特定类型的信纸包裹库存存储在 `stationery:<stationery id>:inventory` 键模式中, 键对应的值是简单的整数, 用来表示那种类型的信纸的可用包裹总数。下面使用 `redis-cli` 命令展示 250 件包裹的初始库存该如何设置:

```
127.0.0.1:6379> SET stationery:1:inventory 250
```

当包裹销售出去后, `stationery:1:inventory` 键对应的整数将被减去销售的包裹数量。同理, 经销商送来新的信纸包裹时, 键将被加上新的信纸包裹总数。

```
127.0.0.1:6379> DECR stationery:1:inventory
```

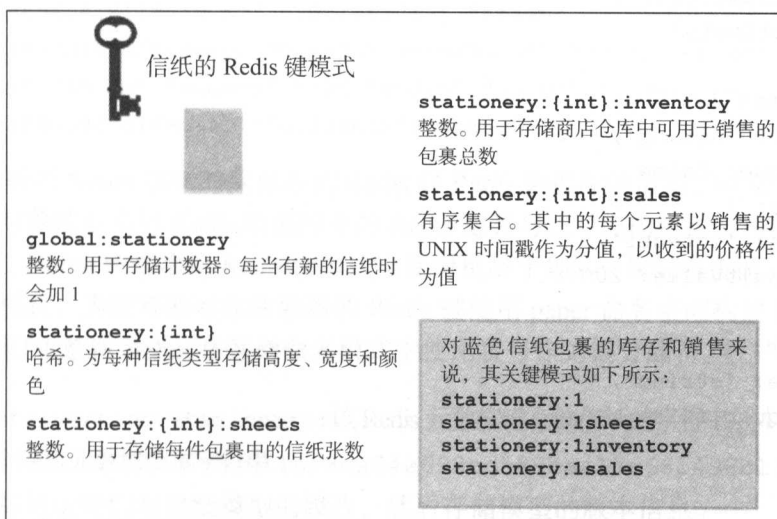
```
127.0.0.1:6379> INCRBY stationery:1:inventory 10
```

每件包裹的销售数据存储在一个有序集合中。集合中的每行以 UNIX 时间戳 (以整数表示自 epoch 以来的时间) 作为分值, 并以销售数量作为值。我们将 `stationery:1:sales` 作为有序集合的 Redis 键, 从 `redis-cli` 上记录一笔 20 美元的销售, 如下所示:

```
127.0.0.1:6379> ZADD stationery:1:sales 1430861194 20.00
```

即使是如此简单的示例, 拥有 Redis 键的共同模式也能为基于 Redis 的应用程序提供将信纸在线商店中的数据关联起来的方法。

下图展示了每种信纸类型聚集在一起的方式及如何使用 Redis 从数据库中快速获取销售数据和其他信息。



解构 Redis 对象映射器

Redis 丰富的生态系统为 Redis 提供了许多对象映射器，它使得键命名管理对设计者和用户透明，同时通过客户端代码，提供类似其他数据存储技术的功能。研究 Redis 对象映射器是如何实现特殊的键模式和数据结构，有助于学习现存的模式并允许你扩展并改进基于 Redis 的应用程序。如果不想重新实现已经存在的功能并在产品环境中运行它，使用 Redis 对象映射器也是有帮助的。一些更为流行的编程语言拥有这些对象映射器项目，用来提供持久化对象语义及 Redis 中的数据的方法，同时为那些可能更为熟悉这些技术和想法的开发者，以他们首选的编程语言提供更多面向对象的方法和技术。这些对象映射器通过使用开发对象映射器的编程语言系统命名法和约定来操作 Redis 的键与值，同时有望减少对组织中基于 Redis 的解决方案的维护与培训的开销。

对 node.js 来说，Redis 对象映射器称为 **Nohm**（详情请参考 <https://github.com/maritz/nohm/>），通过使用 JavaScript 对象模型创建 Redis 模式。回到之前的纸制品网上商店示例中，使用 Nohm 为信纸实体建模，首先需要定义信纸的 JavaScript 模型的颜色、高度和宽度属性，代码如下：

```
nohm.model('Stationary', {
  properties: {
    color: {
      type: 'string',
      unique: false,
      validations: [
        'notEmpty'
      ]
    },
    height: {
      type: 'string',
      unique: false
    },
    sheets: {
      type: 'integer',
      defaultValue: 20
    },
    width: {
      type: 'string',
      unique: false
    }
  }
});
```

创建等效于之前章节中 `stationery:1` 哈希的信纸对象，会产生下列 Redis 命令。同时，Redis 数据库中新的信纸 JavaScript 对象的颜色、宽度和高度值会被设置得和 `stationery:1` 一样。从 `redis-cli` 程序运行 MONITOR 命令会产生以下输出：

```
1431204654.386408 [0 127.0.0.1:61217] "info"
1431204654.404005 [0 127.0.0.1:61217] "get"
"paper:meta:version:Stationary"
1431204654.405394 [0 127.0.0.1:61217] "sismember"
"paper:idsets:Stationary" "-1431204204839"
1431204654.406943 [0 127.0.0.1:61217] "set"
"paper:meta:version:Stationary"
"1bf8ca04e698cd589baa17c661498b1109f8d65c"
1431204654.407516 [0 127.0.0.1:61217] "set"
"paper:meta:idGenerator:Stationary" "default"
1431204654.407547 [0 127.0.0.1:61217] "set"
"paper:meta:properties:Stationary"
"{\"color\":{\"type\":\"string\",\"unique\":false,\"validations\":{\"notEmpty\"}},\"height\":{\"type\":\"string\",\"unique\":false},\"sheets\":{\"type\":\"integer\",\"defaultValue\":20},\"width\":{\"type\":\"string\",\"unique\":false}}"
1431204654.411575 [0 127.0.0.1:61217] "sadd"
"paper:idsets:Stationary" "i9hiar0q75vit5d9rgc5"
1431204654.418524 [0 127.0.0.1:61217] "MULTI"
1431204654.419119 [0 127.0.0.1:61217] "hmset"
"paper:hash:Stationary:i9hiar0q75vit5d9rgc5" "color" "blue"
"height" "40 cm" "sheets" "20" "width" "30 cm" "__meta_version"
"1bf8ca04e698cd589baa17c661498b1109f8d65c"
```

我们将解析当 Nohm 信纸对象保存到 Redis 时 Redis 数据库的活动，研究每个 Redis 键的来源，以及对象映射器对 Redis 数据库中的 Redis 键和与之对应的数据结构做了什么。通过这样的分析，Nohm 所采用的 Redis 键模式会变得更易于理解。我们将遵循一个非常普通的 Redis 设计模式，为所有的对象映射器的 Redis 键使用 `paper` 命名空间构建 Redis 命名模式。同时，我们注意到 Nohm 在基础命名模式中使用冒号作为键分隔符。

- `paper:meta:version:Stationary`: 该 Redis 元数据键存储信纸的字符串版本。该键的值被设置为一个随机元数据版本字符串 `1bf8ca04e698cd589baa17c661498b1109f8d65c`。Nohm 跟踪每次我们对信纸模型的修改，然后存储模型的版本信息。

- `paper:idsets:Stationary`: 该 Redis 集合存储了所有信纸 ID。该集合首先被一个负的 UNIX 时间戳检测, 然后产生了一个值为 `i9hiar0q75vit5d9rgc5` 的 ID 字符串, 并被添加到该集合中。该集合是用来追踪信纸对象的, 随机值可以最小化重复键的问题。
- `paper:meta:idGenerator:Stationary:Nohm` 使用该 Redis 字符串决定生成 ID 的方法。默认的选项产生随机字符串。递增选项则使用整数计数器。
- `paper:meta:properties:Stationary`: 该 Redis 字符串存储了信纸对象的序列化 JSON 元数据。
- `paper:hash:Stationary:i9hiar0q75vit5d9rgc5`: 信纸 Javascript 对象把 `i9hiar0q75vit5d9rgc5` 作为 Redis 键的末尾部分, 将其属性值存储在 Redis 哈希中。这些操作封装在一个事务中。

下一步, 添加第二个信纸包裹, 即红色方形, 45 cm 高×45 cm 宽, 初始纸张数为 15, 因而在数据库中会有下列 Redis 键:

```
paper:meta:properties:Stationary
paper:meta:idGenerator:Stationary
paper:idsets:Stationary
paper:hash:Stationary:i9hjsdjv4o9csf8eeonj
paper:meta:version:Stationary
paper:hash:Stationary:i9hiar0q75vit5d9rgc5
```

我们将看到一个更复杂的 Redis 键模式是如何起作用的。使用 Nohm 为信纸项目的销售建模时, 要使用来自于 `schema.org` 元数据词汇表的两个支持类, 一个名为 `offer` (<http://schema.org/Offer>) 类, 另一个名为 `order` (<http://schema.org/Order>) 类。`schema.org` 词汇表是由 Google、Microsoft、Yahoo 和 Yandex 共同主办的, 用来表示 Web 上的结构化数据。`Offer` 类包含了价格和可用库存, 以及用于支持其他货币的 `priceCurrency` 属性。现在, `priceCurrency` 的默认货币为美元。我们的 `Order` 类包含了 `acceptedOffer` 和 `orderDate` 属性, 其中 `acceptedOffer` 属性连接到我们为信纸创建具体订单。到目前为止, 我们只是使用 Nohm 复制了每个信纸包装的初始存储。添加两个新的模型表示销售数据, 分别命名为 `offer` 和 `order`, 我们想要使用 Nohm 提供的关系建模方法关联到信纸对象。不同于其他基于 SQL 的数据库的对象映射器 (需要在使用之前预先定义), Nohm 允许任何模型通过关联方法关联到其他模型。

```

nohm.model('Offer', {
  properties: {
    inventoryLevel: {
      type: 'integer',
      unique: false
    },
    price: {
      type: 'float',
      unique: false
    },
    priceCurrency: {
      type: 'string',
      unique: false,
      defaultValue: 'USD'
    }
  }
});

```

order 类包含两个属性，分别命名为 orderDate 和 orderedItem。随着纸质文具网上商店的需求变化，order 类可以扩展以便包含其他来自 schema.org 词汇表中的订单属性，例如 customer 和 discount。你可能注意到了我们没有将 orderedItem 添加为 order 类的普通属性，这是因为我们将以 Nohm 关联到 item 信纸的方式来创建 orderedItem。

```

nohm.model('Order', {
  properties: {
    orderDate: {
      type: 'datetime'
    }
  }
});

```

当交易发生时，Nohm 方法会在 offer、order 和 stationery 这三者之间创建关联。在使用交易发生时间创建新的订单实例之后，Nohm 使用几个 Redis 集合为这三个不同的类之间的关系进行建模。Nohm 将关系信息存储在几个不同的集合中，我们可以从 redis-cli 程序中一探究竟。

首先，红色信纸的哈希键为 paper:hash:Offer: ia4ev8iu8cns7w6p968h，并将库存级别属性设置为 50，价格为 15。

```
1432589868.318914 [0 10.0.2.2:55200] "hmset"
"paper:hash:Offer:ia4ev8iu8cns7w6p968h" "inventoryLevel" "50" "price" "15"
"priceCurrency" "USD"
"__meta_version" "
229e1d3b89b02804b4bdad9909fa75aa442197d5"
```

下一步是创建 `paper:relationKeys:Offer:ia4ev8iu8c ns7w6p968h` 和 `paper:relations:Offer:itemOffered:Stationery:ia4ev8iu8cns7w6p968h` 集合。第一个集合中存储的键所对应的集合中，存储了那些通过 `itemOffered` 属性创建 `offer` 和 `stationery` 之间的关联。第二个集合通过创建具体的 `offer` 和 `stationery` 之间的具体关联，存储了所有单独的信纸 ID。

```
1432589868.323265 [0 10.0.2.2:55200] "sadd" "paper:relationKeys:Offer:
ia4ev8iu8cns7w6p968h"
"paper:relations:Offer:itemOffered:Stationery:ia4ev8iu8cns7w6p968h"
1432589868.323281 [0 10.0.2.2:55200] "sadd"
"paper:relations:Offer:itemOffered:Stationery:ia4ev8iu8cns7w6p968h"
ia4ev8itec2wq9gc0qnt"
```

当接收订单并且交易被确认时，首先，Redis `paper:hash:Order:1` 哈希使用订单日期属性被创建出来。同时，通过 `Nohm`，元数据版本 `id` 使用哈希值作为属性被存储。

```
1432589868.325604 [0 10.0.2.2:55200] "hmset" "paper:hash:Order:1"
"orderDate"
"Mon May 25 2015 15:33:33 GMT-0600 (Mountain Daylight Time)"
"__meta_version" "a881a941cb6ff674a79c7f652f8d8153b7b47b"
```

两个额外的集合，分别命名为 `paper:relationKeys:Order:1` 和 `paper:relations:Order:offer:Offer:1`，创建了 `order` 和 `offer` 之间的关联。第一个集合为 `order` 存储了所有的关系连接。第二个集合为之前命令添加的 `order` 存储了特定的 `Offer`。

```
1432589868.327808 [0 10.0.2.2:55200] "sadd"
"paper:relationKeys:Order:1" "paper:
relations:Order:offer:Offer:1"
1432589868.327829 [0 10.0.2.2:55200] "sadd"
"paper:relations:Order:offer:Offer:1" "ia4ev8iu8cns7w6p968h"
```

下图展示了 JavaScript 代码的执行流程，为我们的线上纸制品商店创建了 `stationery`、

offer 和 order 之间的关联。



键过期

Redis 的一个极其重要的特性是能够为键设置过期时间。通过自动化删除过期键，Redis 应用程序能够更好地管理数据库所使用的内存大小和使用情况，同时减少用于追踪数据库中每个键的客户端代码量。

第 3 章针对的是 Redis 实例内存优化和管理，将会更详细深入地探索键过期这一主题。我们经常会这样的背景下讨论键过期——基于可接受的性能限制来维持 Redis 实例的内存使用情况。维持这一性能的方法是从数据库中驱逐（evict）过期的键。Redis 提供了许多不同的模式，可以根据应用程序的需求和性能限制设置过期键的自动驱逐策略。设置的方法可以是对 Redis 配置文件的选项进行设置，也可以在运行时向 Redis 数据库发送命令进行设置。

键的注意事项

多年来，一些最佳实践逐渐浮现，并在 Redis 的 tutorial 里进行了简要阐述。这些实践的焦点围绕着运行中的 Redis 数据库和用于支持的客户端代码的清晰性与性能的权衡。Redis 键的大小应该受到限制，不仅因为键的大小超过 1024 字节会导致内存增长，大尺寸

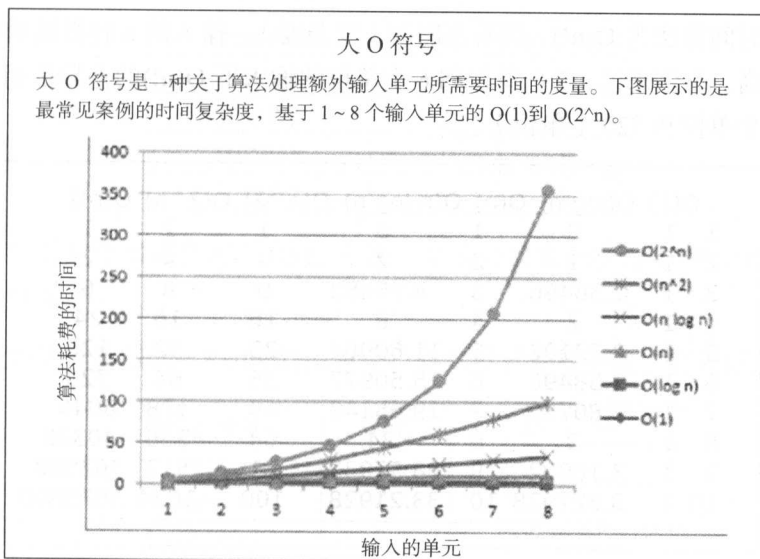
的键还会令 Redis 实例的开发者和用户感到困惑。过长的键名称带来的另一个问题是随着 Redis 实例大小的增长,这些过长的键名称开始消耗更多的内存,从而挤压了正常数据所需的内存空间。

同样,如果键名称太短,额外节省的内存可能得不偿失。因为在对 Redis 进行故障排除或者通过新的 Redis 键添加新的功能时,会碰到各种问题。举例来说,一个名为 `u11:2` 的键虽然比较短,但是并没有传达出值的含义。在数据相同的情况下,键名为 `user:11:clicks` 可以更好地描述存储在 Redis 键上的值,同时也能更好地表达该键的应用程序上下文。随着时间的推移,应用程序不断发展与演化带来了挑战,可以通过采用一致的 Redis 键模式为未来的成长留出空间。在开发 Redis 键模式时,多花点时间思考未来可能的用法,就能缓解对客户端代码的大规模重构,以及为了处理来自个人用户或者其他程序针对应用程序提出的新需求而导致的数据迁移。

Redis 的 `KEYS` 命令应当在万不得已时使用,因为它会对 Redis 实例造成长时间的阻塞,甚至会导致 Redis 内存耗尽。`SCAN` 命令为 Redis 中所有的键提供了一个迭代器,可以对所有的键进行增量式调用。Redis 的 `SCAN`,以及分别针对哈希、集合和有序集合的等效 `HSCAN`、`SSCAN` 和 `ZSCAN` 命令相对较新,符合 Redis 应用程序的实际需求。在使用 `SCAN` 和相关的迭代器命令时需要注意的是,如果一个元素在从头到尾的迭代中不是始终存在,那么 `SCAN` 命令并不保证该元素能够返回。

大 O 符号

正如你可能已经知道的,Salvatore Sanfilippo 有意将每个 Redis 命令的最坏算法性能整理成文放在 Redis 网站上(<http://redis.io/commands/>)。对此,我们由衷感谢。这种对性能上的算法指标的关注,作为核心可操作度量,使得 Redis 区别于其他数据库技术。数学上对大 O 符号的定义为“象征性地表达给定函数的渐进行为”。在计算机科学和对 Redis 中大 O 符号的理解的帮助下,我们能够通过这些命令在面对不断增长的输入时的性能表现,对 Redis 命令做出性能上的区分。



图解大 O 符号

在 Redis 文档中，每个 Redis 命令的时间复杂度由以下大 O 示例给出。

- 大 O 符号中的 $O(1)$ 表示的是随着输入的增长而不会对时间或处理造成变化的情况。在 $O(1)$ 算法中，性能的上限是线性时间，这意味着随着输入的增加不会导致性能的下降，但受算法本身复杂性的限制。
- 下一个大 O 示例为 $O(\log n)$ 或者称为对数时间（复杂度）。它对每个输入进行操作，返回的结果大于 $O(1)$ ，但是性能等价于对 n 求对数。
- 对于大 O 符号中术语 $O(n)$ 的直观理解遵循常识观念，即添加额外的单元将以恒定比例的量增加处理时间。
- $O(n \log n)$ 即对数线性时间， $O(\log n)$ 作用于每一个输入之上。实际上，在 $O(n \log n)$ 算法中每次输入都增加了一倍以上。
- 对于 $O(n^2)$ ，即平方时间来说，随着 n 的增长，时间的量也成倍增长。对于每个加倍的 n 来说，时间处理变为原来的 4 倍。 $O(n^2)$ 算法的性能在 n 较小的情况下也许是可以接受的，但当 n 增长到一定量时，就很快变得不切实际了。
- 就那些需要 $O(2^n)$ 即指数时间（复杂度）来解决问题而言，对于每一个额外的输入，时间都会加倍，这就导致对大多数较大的 n 来说， $O(2^n)$ 变得毫无用处。

- 最复杂的时间算法为 $O(n!)$ ，即阶乘时间（复杂度），输入的 n 轻微地增加都会导致处理时间过高。举例来说，5 个单位和 6 个单位的输入下 $O(n!)$ 的差异是相当大的（分别用时 120 个单位和 720 个单位）。

	$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(2^n)$	$O(n!)$
1	1	0	1	0	1	2	1
2	1	1	2	2	4	4	2
3	1	1.58496	3	4.75488	9	8	6
4	1	2	4	8	16	16	24
5	1	2.32192	5	11.60964	25	32	120
6	1	2.58496	6	15.50977	36	64	720
7	1	2.80735	7	19.65148	49	128	5040
8	1	3	8	24	64	256	40320
9	1	3.16992	9	28.52932	81	512	362880
10	1	3.321928	10	33.21928	100	1024	3628800

为自定义代码计算大 O 符号

根据 Redis 文档中为每个命令提供的大 O 符号，我们可以为任何提议的基于 Redis 的解决方案计算出一个粗略的效率估计。一个简单的方案是对于一定级别的 n 将所有 Redis 命令的大 O 符号相加，然后为实现代码估计大 O 符号，以便为整个解决方案做粗略的时间效率估计。举例来说，用 Redis 实现的缓存只是简单的单个 SET 和 GET 调用，该解决方案的大 O 符号就是 $O(1) + O(1) \approx 2$ 个时间单位。更为复杂的用例需要更多的命令及更高级的大 O 符号。

对数据结构的时间复杂度的评估不仅包括数据结构本身，还包括对数据进行采集与提取等 Redis 命令总数的优化。使用场景不同，不可一概而论。例如，将数据采集到有序集合中耗费较多的执行时间，同时能在低于大 O 示例的情况下访问数据，有时是完全可以接受。另一个相反的用例场景是，以低延迟的方法采集大量的数据，需要尽量快的采集速度（低于大 O 示例），而同时对于快速访问这些数据的需求不是很强烈。比方说，当存储那些会在一定时间内过期的日志信息时，虽然优化访问很重要，但在这一场景中，优化写比访问更重要。

那么，回到信纸示例中。让我们比较一下我们的第一个自定义 Redis 解决方案和第二个基于 Nohm 的解决方案。由于没有第一个解决方案的客户端代码，因此我们只比较 Redis 命令。我们首先研究的是添加蓝色和红色信纸到 Redis 数据库的总时间复杂度。

Redis 命令	O(n)	总 计
INCR	+1	1
HMSET	+3	4
INCRBY	+1	5
HMSET	+3	8

将一个信纸对象添加到自定义 Redis 解决方案的总共复杂度为 4, 算上额外的红色信纸包裹的话, 总计是 8。

现在, 我们来分析 Nohm Redis 监控命令, 并对设置和保存蓝、红信纸进行统计:

Redis 命令	O(n)	总 计
INFO	+1	1
GET	+1	2
SISMEMBER	+1	3
SET	+1	4
SET	+1	5
SET	+1	6
SADD	+1	7
HMSET	+3	10
SADD	+1	11
HMSET	+3	14

对 Nohm 解决方案来说, 添加一个信纸对象包含多个设置命令, 总共的复杂度为 10。与自定义 Redis 解决方案类似, 算上额外的红色信纸包裹的话, 一共是 14。依据应用程序模型, 选用的编程语言所支持的 Redis 对象映射器可能会随着应用程序的扩展, 带来相对较小的负担。理想情况下, 我们将客户端的时间复杂度控制在 $O(n \log n)$ 以下, 目标是到 $O(1)$ 或者 $O(\log n)$ 。当应用程序随着时间逐渐成熟, 这种想法会变得越来越困难, 通过边界案例和来自终端用户的反馈帮助你探索和工作吧。

虽然这两个 Redis 解决方案间总结得出的时间复杂度差异相对较小, 但是 Nohm Redis 对象映射器提供了许多基本功能。如果我们想使用 Redis 作为数据库构建完整的 node.js 应用程序, 这些功能都是我们需要复制过来的。再仔细想想, 我们需要的是自定义 Redis 解决方案提供的极快的速度但是有限的对象追踪和验证支持功能, 还是 Nohm 提供给应用程序额外的对象元数据和字段验证功能, 这都需要去权衡。

回顾 Redis 数据结构的时间复杂度

基于对计算大 O 符号的理解,接下来简要回顾 Redis 的基本数据结构。需要注意的是,采用 Redis 当前支持的命令操作数据结构的时间复杂度所带来的启示。

字符串

Redis 值中最基本的数据结构为字符串,也就是和 Redis 键相同的数据类型。使用 Redis 最简单的方式就是字符串对字符串的键值存储。注意,Redis 有着和其他诸如 Memcached 之类键值数据存储解决方案相似的性能特点。

在 Redis 中,字符串并不仅仅是那些高级编程语言中包含字母数字字符的字符串,而是包含 C 语言(Redis 主要采用的编程语言)的序列化字符。Redis 字符串中最基础的 GET 和 SET 命令是 $O(1)$ 操作。这使得 Redis 作为简单的键值存储极其快速。在思考 Redis 解决方案时,GET 和 SET 命令使用起来的快速和简单不容忽视。在我的经验中,我总是过早地想当然地使用最复杂的 Redis 数据结构,例如有序集合。对于解决眼前的问题来说,简单的 Redis 字符串可能是更快速、更简单的方法。

对于大多数 Redis 字符串操作来说,访问和采集命令的时间复杂度要么是 $O(1)$,要么是 $O(n)$ 。其中 $O(n)$ 字符串命令大多是块命令,例如 GETRANGE、MSET 和 MGET。GETRANGE 命令是一种 $O(n)$ 操作,其中 n 为返回字符串的长度。如果将该操作比作一系列小的 GET 命令(虽然 GET 不返回存储在键中的字符串的子串),则理解起来更直观。我们将利用 redis-cli 演示 SET 和 GETRANGE:

```
127.0.0.1:6379> SET organization:1 "The British Library"
127.0.0.1:6379> GETRANGE organization:1 4 10 "British"
```

因此,在该示例中,对于 SET 命令来说大 O 符号+1 同时对于 GETRANGE 来说大 O 符号+6,等价于发送独立的伪 GET 命令获取 6 个字符。

由于 Redis 将所有数据作为字符串存储,特定字符串的类型信息也会被维护起来以支持 INCR/DECR 和 bitstring 命令。对于 INCR 和 DECR 命令来说,存储的值是以 10 为基数的 64 位有符号整数字符串,如果该值被其他诸如 APPEND 的 Redis 字符串命令修改过,可能会导致损坏。因而之后作用在同一键上的与整数相关的 Redis 命令都会失败。我们可以从 redis-cli 上简单重现该场景,使用下列 INCR、GET 和 DUMP 命令作用在 new:counter 键上:

```

127.0.0.1:6379> INCR new:counter(integer) 1
127.0.0.1:6379> GET new:counter
"1"
127.0.0.1:6379> DUMP new:counter
"\x00\xc0\x01\x06\x00\xb0\x95\x8f6$T-o"
127.0.0.1:6379> APPEND new:counter "a"
(integer) 2
127.0.0.1:6379> INCR new:counter
(error) ERR value is not an integer or out of range
127.0.0.1:6379> GET new:counter
"1a"
127.0.0.1:6379> DUMP new:counter
"\x00\x021a\x06\x00\x8br\x9a\x98-9\x9a\xa6"

```

哈希

哈希，就如在其他编程语言中的字典或者关联数组那样，是一种将一个或多个字段映射到对应的值的数据结构。在 Redis 中，所有的哈希值必须是 Redis 字符串，并且有唯一的字段名。字段的值是简单的 Redis 字符串。通过调用 Redis 的 HGET 或 HMGET 命令，同时传入合适的 Redis 键和一到多个字段参数，就能返回字段的值。对于大多数使用场景来说，Redis 哈希为 HSET 和 HGET 命令提供了很棒的 $O(1)$ 性能。与字符串块命令类似，哈希的 HGETALL、HMSET、HMGET、HKEYS 和 HVALS 命令均为 $O(n)$ 。如果哈希非常小，那么返回所有哈希键和值的 HGETALL 和 HMGET 命令之间没有十分明显的差异。当哈希中键和值不断增长时，两者之间的差异可以让应用程序大不相同。假设哈希中有 1 000 个字段，如果你的应用程序只是经常使用其中的 300 个，对 Redis 调用 HGETALL 或 HVALS 的时间复杂度为 $O(1000)$ ，而使用 HMGET 的时间复杂度只有 $O(300)$ 。这是因为虽然 HGETALL 和 HMGET 都是 $O(n)$ ，但是对于 HMGET 命令来说，其上限为所请求字段的总数而非整个哈希。哈希的总体较小时，将 HMGET 替换为 HGETALL 命令是增加 Redis 程序性能的一种方式。对于大型哈希来说，返回大量值的 HMGET 命令在完成执行前会阻塞其他客户端接收数据，从而极大地影响 Redis 的总体性能。在这种情况下，有针对性的 HGET 会是个更好的选择。

虽然 Redis 哈希的值不能包含哈希、列表或者其他数据集合结构，但是 Redis 提供了 HINCRBY 和 HINCRBYFLOAT 命令，允许你将字段中存储的字符串值当作整数或者浮点数操作。在接下来的例子中我们可以从 Redis 命令行上看到，如果你尝试更新字段的值但是弄错了数据类型，Redis 会返回错误：

```

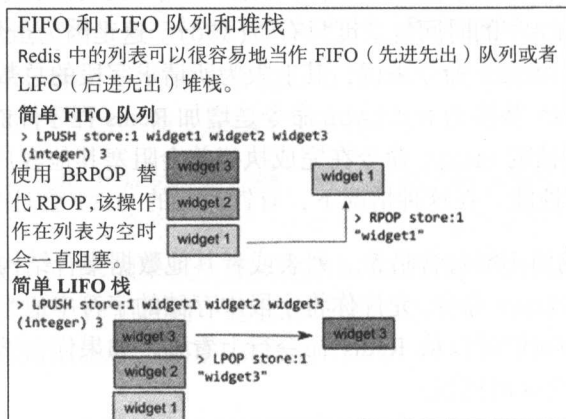
127.0.0.1:6379> HMSET weather:2 temperature 46 moisture .001
127.0.0.1:6379> HINCRBY weather:2 temperature -1
127.0.0.1:6379> HGET weather:2 temperature
"45"
127.0.0.1:6379> HINCRBY weather:2 moisture 1
(error) ERR hash value is not an integer
127.0.0.1:6379> HINCRBYFLOAT weather:2 moisture 1
127.0.0.1:6379> HGET weather:2 moisture
"1.001"

```

Redis 会根据命令来区分设置的值 1 是整数还是浮点数。

列表

在 Redis 中列表是字符串的有序集合，它允许重复的字符串值。Redis 中的列表被更准确地标记和实现为链表。由于 Redis 列表以链表的方式实现，使用 LPUSH 向列表前端或者使用 RPUSH 向列表末尾添加条目是相对廉价的操作，表现为常数时间复杂度 $O(1)$ 。对 LINSERT 和 LSET 命令来说，时间复杂度是线性的 $O(n)$ ，但两者有些重要的差别。对 LSET 命令来说，你可以指定下标值来设置列表的值，由于本质上是链表，因此变量 n 是列表的长度，同时不管是设置列表中的第一项还是最后一项，时间复杂度均为 $O(1)$ 。对 LINSERT 命令来说，你可以在参考值之前或者之后插入值，上述操作的时间复杂度为 $O(n)$ 。其中 n 为列表元素的个数。该命令必须一直查找直到获取到参考值，最坏的情况是将值插入列表的末尾。记住，在用大 O 符号时，我们感兴趣的是最坏场景，因此 LINSERT 命令的时间复杂度被认为是 $O(n)$ ，即便是特殊情况下参考值是列表当中第一个元素，使得 LINSERT 命令的复杂度为 $O(1)$ 。



对 LRANGE 命令来说,官方 Redis 文档给出的复杂度分类为 $O(s+n)$, 其中 s 为从列表的表头或者表尾到偏移量位置的元素个数,这取决于列表的大小。 n 代表返回的元素总数。如果你想要返回整个列表,常见的操作模式为 LRANGE mylist 0-1。因此,对于长度为 10 的列表来说,该操作的时间复杂度为 $O(10+10)$ 。通常,LTRIM 命令的时间复杂度为 $O(n)$, 其中 n 为返回给客户端的元素数量。正如官方文档中提到的有关 LTRIM 命令的说明那样,使用 LTRIM 结合 RPUSH 或者 LPUSH,是存储固定长度的集合的常用方法。举例来说,如果只想保存最近 7 天有价值的平均温度数据,请使用下列 Redis 命令:

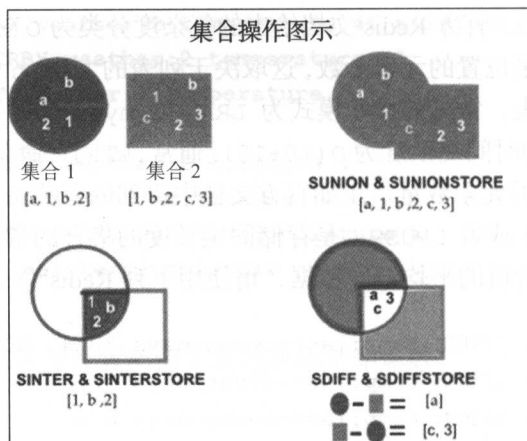
```
127.0.0.1:6379> LPUSH temp:last-seven-days 30 45 50 52 49 55 51
127.0.0.1:6379> LPUSH temp:last-seven-days 56
127.0.0.1:6379> LTRIM temp:last-seven-days 0 6
127.0.0.1:6379> LRANGE temp:last-seven-days 0 -1
1) "56"
2) "51"
3) "55"
4) "49"
5) "52"
6) "50"
7) "45"
```

正如我们看到的那样,这种模式允许我们存储最近 7 天的平均温度,并且当采用这种方法时,由于每次只有一个值添加到列表中,LTRIM 命令的时间复杂度现在接近 $O(1)$ 了。

集合

Redis 中的集合保证了字符串值的唯一性,但是不保证这些值的顺序。Redis 集合实现了集合语义的并集、交集和差集,并在 Redis 实例中将这些集合操作的结果存储为一个新的 Redis 集合。以当前的 Redis 集群的实现来说,并集、交集和差集这些集合语义受到了诸多限制,并且只能以受限的方式使用。SADD 命令将一到多个值添加到集合中,时间复杂度为 $O(n)$, 其中 n 是需要添加到集合的元素总数。重要的 SISMEMBER 命令用于判断值是否为集合的成员,时间复杂度为 $O(1)$ 。SMEMBERS 命令返回集合中所有成员的列表,时间复杂度为 $O(N)$ 。集合可能有着与 Redis 中其他数据结构相似的性能。在某些情况下,相较于哈希,集合拥有更佳的内存使用率。

Redis 中集合特别有用的地方在于对集合并集、交集和差集操作的支持,所有这些操作有着不同的时间复杂度,在 Redis 集群中使用时会受到限制。



SUNION 和 SUNIONSTORE 命令允许将多个集合的所有成员返回给客户端或者存储为 Redis 中的新集合。这两个命令的时间复杂度均为 $O(n)$ ，其中 n 为所有集合中元素的总数。SINTER 和 SINTERSTORE 命令返回集合的交集，后者会将返回的集合存储在 Redis 中。在 Redis 中，多个集合的公共成员的计算时间复杂度为 $O(n*m)$ ，其中 n 是最小集合的大小， m 为集合的总数。最后，SDIFF 和 SDIFFSTORE 命令将返回或者存储第一个集合和后续集合之间的差异。和 UNION 命令相同，SDIFF 和 SDIFFSTORE 命令的时间复杂度为 $O(n)$ ， n 为所有集合的元素总数。Redis 集合的效用就体现在这些集合操作上，允许在非常基础的布尔逻辑层面操作数据。但是，随着集合大小的增长，数据处理的时间也会以最小的 $O(n)$ 时间复杂度增长。

有序集合

在 Redis 中，有序集合数据类型兼具 Redis 列表和集合的特性。与 Redis 列表相似，有序集合中的值是有序的，同时与集合类似，有序集合中的每个值都是唯一的。在 Redis 的所有众多数据结构中，有序集合是最接近杀手级的功能。有序集合的灵活性允许根据应用程序的需求采用不同的使用模式。在游戏中使用单一有序集合可以记录玩家得分，可以通过 ZRANGE 或 ZREVRANGE 命令，从排行榜中获取排名靠前和排名靠后的玩家。

对有序集合来说，ZADD 命令将成员和分值一起添加到有序集合中。ZADD 命令的时间复杂度为 $O(\log(n))$ ，意味着随着有序集合的大小的增加，处理时间的增加比率是一个常量。因此，向大型有序集合中插入更多的新值之间的差异是微不足道的，例如以下两者 $\log(10000) \sim 9.21034037$ 和 $\log(10001) \sim 9.21044036$ 的差异仅为 .000099。

另一个 Redis 有序集合非常受欢迎的功能是，如果有序集合中所有或者部分元素的分值相同，这些值将以字典字母顺序进行排序。该功能可以方便地用于文本字符串的字母排序。我们可以通过以下方式演示该功能：首先，将 7 种颜色添加到一个名为 `colors` 的有序集合中：

```
127.0.0.1:6379> ZADD colors 0 red 0 blue 0 green 0 orange 0 yellow 0  
purple 0 pink
```

现在，通过 `ZRANGE` 命令将颜色以字母顺序取出：

```
127.0.0.1:6379> ZRANGE colors 0 -1
```

```
1) "blue"  
2) "green"  
3) "orange"  
4) "pink"  
5) "purple"  
6) "red"  
7) "yellow"
```

可以使用 `ZREVRANGE` 命令，以字母倒叙的形式获取数据：

```
127.0.0.1:6379> ZREVRANGE colors 0 -1
```

```
1) "yellow"  
2) "red"  
3) "purple"  
4) "pink"  
5) "orange"  
6) "green"  
7) "blue"
```

不管哪个示例，在 `colors` 有序集合中，所有的分值都是相同的。我们可以使用 `ZREVRANGE` 命令并带上 `WITHSCORES` 关键字，如下所示：

```
127.0.0.1:6379> ZREVRANGE colors 0 -1 WITHSCORES
```

```
1) "yellow"  
2) "0"  
3) "red"  
4) "0"  
5) "purple"  
6) "0"  
7) "pink"
```

- 8) "0"
- 9) "orange"
- 10) "0"
- 11) "green"
- 12) "0"
- 13) "blue"
- 14) "0"

Redis 有序集合中的字母符号

ZRANGEBYLEX、ZLEXCOUNT 和 ZREVRANGEBYLEX 这三个 Redis 命令都有表示区间的特殊语法，用于获取有序集合中拥有相同分值的项。

```
127.0.0.1:6379> ZADD letters 0 a 0 b 0 c 0 d 0 e 0 f
127.0.0.1:6379> ZRANGEBYLEX letters (a +
1) "b"
2) "c"
3) "d"
4) "e"
5) "f"
127.0.0.1:6379> ZRANGEBYLEX letters [b +
1) "b"
2) "c"
3) "d"
4) "e"
5) "f"
127.0.0.1:6379> ZRANGEBYLEX letters [b (e
1) "b"
2) "c"
3) "d"
```

(
用于排除范围项

[
用于包含范围项

+
表示正无穷

-
表示负无穷

Redis 同时提供了 2.8 版本引入的专门的命令 LRANGEBYLEX 和 LREVRANGEBYLEX，以字典顺序获取元素。这些命令允许你通过特定的语法指定有序集合中的起止位置。在字符前的 "(" 将排除该值，而 "[" 将包含该值。同时，"+" 是正无穷的缩写，"-" 是负无穷的缩写。下列命令使用之前的 colors 有序集合，有助于展示差异：

```
127.0.0.1:6379> ZRANGEBYLEX colors (b [p
1) "blue"
2) "green"
3) "orange"
127.0.0.1:6379> ZRANGEBYLEX colors - +
1) "blue"
2) "green"
3) "orange"
4) "pink"
5) "purple"
```

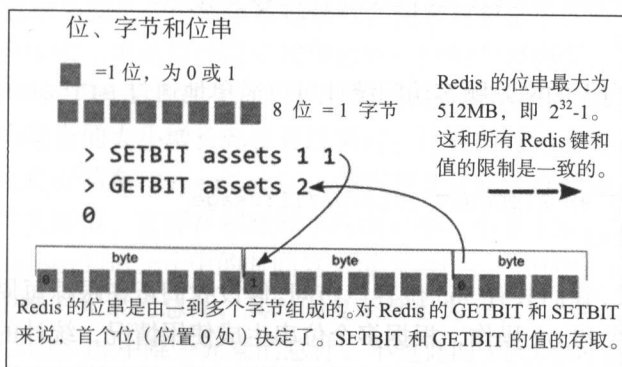
- 6) "red"
- 7) "yellow"

高级有序集合操作

与集合类似，Redis 中的有序集合支持并集和交集的集合操作，虽然有序集合中这些操作的时间复杂度要比集合的差。有序集合操作的另一个问题在于，当使用 Redis 集群时，并集和交集的操作只能用在有序集合键被分片到同一个哈希槽中，且运行在同一个节点上的情况下。Redis 命令 `ZINTERSTORE` 的时间复杂度为 $O(nk) + O(m \log(m))$ ， n 为最小有序集合的大小， k 为这些求交集的有序集合的总数， m 为最后返回的有序集合计算结果中元素的个数。同样地，`ZUNIONSTORE` 命令的时间复杂度为 $O(n) + O(M \log(M))$ ， n 代表所有有序集合的大小总数， m 代表最终有序集合的元素总数。考虑到有序集合的这些特性，以上两个集合操作所需要的额外时间是可以接受的。大型集合和有序集合之间的性能差异与是否需要排序无关，记住这一点很有用。

位串和位操作

Redis 字符串和对应命令的特殊用法允许为 Redis 中相对较少数量的比特使用内存高效的数据结构。同时，取决于具体的用例场景和数据，使用集合和哈希会提供更好的性能。在位串中，每个字节存储 8 位，其中位置 0 处为最高有效位，它被设置成 0 或者 1。Redis 的位串最大为 512MB，这和 Redis 所有的键和值的限制是一致的。



位串如此高效快速的一个原因是大多数针对它的命令的时间复杂度为 $O(1)$ 或 $O(n)$ 。使用 `SETBIT` 和 `GETBIT` 命令，可以将位设置为 0 或 1，或者获取值，这两种操作的时间

复杂度均为 $O(1)$ 。位串对于存储一系列连续值的二进制信息来说极其迅速。对 BITOP、BITPOS 和 BITCOUNT 这些命令来说,虽然时间复杂度为 $O(n)$,但是对位串的使用提供了强大的语义。

位串常用的用例场景是存储用于表示一系列顺序的键的布尔值,即 0 或者 1。正如首次的博客中提到的那样,使用 Redis 创建业务仪表盘,每月、每天,甚至每小时的使用信息的存储能够以非常有效的方式通过使用通用的 Redis 键完成。举例来说,如果想在网站上追踪每日的使用情况,可以从简单的“customer:”模式开始,用来为每位客户存储用户名、哈希过的密码和电子邮件地址,如下所示:

```
127.0.0.1:6379> INCR global:customer
2445
127.0.0.1:6379> HMSET customer:2445 username mmaxwell password
'49fdb34f64be0a29af77ae77370a77232c3d6c37' email
mmaxwell@gmail.com
```

假设顾客计数从 0 开始,顾客 mmaxwell 是连续第 2445 位顾客。现在,为了记录 mmaxwell 在 2016 年 2 月 11 日访问了我们的网站,我们会设置 2016/02/11:usage 位串的第 2445 位,如下所示:

```
127.0.0.1:6379> SETBIT 2016/02/11:usage 2445 1
```

如果我们想查看顾客 mmaxwell 是否在那天访问了我们的网站,可以通过 GETBIT 命令获取存储在 2445 上位值,如下所示:

```
127.0.0.1:6379> GETBIT 2016/02/11:usage 2445
(integer) 1
```

查找 2 月 11 日当天的网站顾客访问统计可以简单地通过 BITCOUNT 命令达成,如下所示:

```
127.0.0.1:6379> BITCOUNT 2016/02/11:usage
(integer) 365
```

在 2 月的这一天,我们共有 365 位客户访问。假设我们正在跟踪每周顾客的使用情况,可以使用 BITOP 命令和 OR 操作,根据多个位串生成使用情况,统计的结果存储在新的键中,如下所示:

```
127.0.0.1:6379> BITOP OR 2016/02/week2:usage 2016/02/07:usage
```

```
2016/02/08:usage 2016/02/09:usage 2016/02/10:usage 2016/02/11:usage
2016/02/12:usage 2016/02/13:usage
127.0.0.1:6379> BITCOUNT 2016/02/week2:usage
(integer) 1834
```

为了计算月度总计并将结果存储在新的键 2016/02:usage 中，可以再次执行 BITOP 命令，通过 4 周的使用情况键来计算总和：

```
127.0.0.1:6379> BITOP OR 2016/02:usage 2016/02/week1:usage 2016/02/
week2:usage 2016/02/week3:usage 2016/02/week4:usage
(integer) 306
127.0.0.1:6379> BITCOUNT 2016/02:usage
(integer) 6893
```

最后，整个网站的年度总计可以针对 12 个月的位串再次调用 BITOP OR 操作：

```
127.0.0.1:6379> BITOP OR 2016:usage 2016/01:usage 2016/02:usage
2016/03:usage 2016/04:usage 2016/05:usage 2016/06:usage 2016/07:usage
2016/08:usage 2016/09:usage 2016/10:usage 2016/11:usage 2016/12:usage
(integer) 306
127.0.0.1:6379> BITCOUNT 2016:usage
(integer) 73190
```

HyperLogLogs

最新的 Redis 数据类型是一个概率数据结构，用来对集合中的唯一项做估计总数。通常情况下，获取集合中唯一项的总计需要和项总数一样的大量内存，并且时间复杂度至少为 $O(n)$ 。为什么呢？为了确保没有项被重复计算，算法必须为每个项保存一条记录，用于匹配其他新的项。当集合的大小增长到百万级别时，计算的开销变得非常巨大而且昂贵。相比之下，将唯一元素存储在 HyperLogLog 结构中会计算并存储集合大小的估算（可以看成是概率）以取代实际值，它存在相对较小的误差率，小于 1%。使用 PFADD 命令将一到多个元素添加到 HyperLogLog 中的时间复杂度为 $O(1)$ ，而通过 PFCOUNT 命令获取单个 HyperLogLog 中唯一元素的总计，其时间复杂度也为 $O(1)$ 。通过 PFCOUNT 命令，可以计算多个 HyperLogLogs 中唯一元素的总计，不过此时 PFCOUNT 的性能为 $O(n)$ ， n 代表键的总数。

为了见证集合和 HyperLogLog 两者在性能上的差异，考虑以下示例。假设你有超过

50,000 个唯一顾客不断地被企业 CRM 系统添加和减去，可以通过以下命令实现将 50,000 个客户的键存储到集合中：

```
127.0.0.1:6379> SADD customers-set customer:1 customer:2 customer:3 ...  
customer:52111  
127.0.0.1:6379> SCARD customers-set  
(integer) 52411
```

使用 HyperLogLog 执行相同的操作，如下所示：

```
127.0.0.1:6379> PFADD customers-hll customer:1 customer:2 customer:3  
... customer:52111  
127.0.0.1:6379> PFCOUNT customers-hll  
(integer) 52213
```

我们可以观察到在该示例中，HyperLogLog 估算的 52213 和 Redis 集合的实际总计 52411 之间的差异为 198，换算成百分比为 0.4%。这低于最坏情况下 HyperLogLog 的总数估算值 1%。随着数据集大小的不同，得到的结果也不同。如果不需要精确的唯一元素总数，并且对应用程序来说非常接近的估算就足够，那么 HyperLogLog 将正和你意。

总结

任何 Redis 应用程序都有两个关键部分，即键和这些键上存储的值。对大多数 Redis 解决方案来说，键的命名设计至关重要，不管是手工设计模式还是使用 Redis 对象映射器将细节隐藏在客户端层的抽象背后。所有 Redis 数据结构的性能，以及对应的写和访问命令，都以大 O 符号的形式进行了评估。该方法用于在计算机科学中计算给定增长的输入的算法在最坏情况下的性能。使用大 O 符号，我们可以通过总结客户端代码中的函数、方法或者类中的所有 Redis 命令的性能，估算基于 Redis 的应用程序的效用。接下来，我们对 Redis 的字符串、哈希、列表和集合进行基本的复杂度分析。同时，我们也阐述了一些有关 Redis 有序集合、位串和 HyperLogLogs 数据结构的高级使用方法。

在第 3 章里，我们将研究 Redis 项目关键的一面，即对 Redis 数据库可用内存的优化、改进与管理。我们将详细阐述 Redis 键模式的构造是如何对内存的使用造成正负两面的影响的。同时，我们也将阐述 Redis 中可用的多种键过期选项，以及 Redis 不同的缓存策略，包括最近最少使用（LRU）是如何帮助 Redis 数据库在受环境约束的情况下保持一定大小的。

3

内存管理的建议与技巧

相较于大多数数据存储技术，想要高效地使用 Redis 需要充分理解计算机的内存技术及网络和硬盘延迟，以便追踪性能瓶颈、处理资源规划及分配。Redis 将所有数据都加载到 RAM 时，应用程序的读写受到硬件和网络连接的技术限制，然后才受到诸如 Oracle 或 MySQL 等传统关系型数据库所使用的更慢的磁盘读写操作限制。就像第 2 章中所讨论的那样，软件的时间复杂度及它与 Redis 的交互方式作为优化设计的目标来说更为重要。本章将先回顾一些用来配置 Redis 的内存相关指令。这些指令可以在 `redis.conf` 文件中进行设置。

接下来着重讲解 Redis 的内存优化。首先是使用 Redis 的主从复制时需要做出的内存考虑。之后讨论的主题可能有违直觉：使用 32 位版本的 Redis 服务器使内存最大化。在研究了 Redis 的键过期选项后，我们将学习关于不同的驱逐（evict）键策略。Redis 能使用这些策略处理内存耗尽的关键场景，其中受欢迎的方法是最近较少使用（**Less Recently Used**, LRU）缓存算法来驱逐键。在 LRU 之后，我们将对特殊的、内存高效的数据结构进行研究。这些数据结构被 Redis 用来实现相对较小的哈希、列表、集合及排序集合。恰好借此讨论在基于 Redis 的应用中，哈希所采用的内存节约的键值使用模式。本章最后将简单讨论有关硬件和网络延迟的问题，以及 Redis 是如何通过调整内存使用来弥补这些延迟的。

配置 Redis

如果想要运行一个内存高效的 Redis 数据库，首先需要理解那些在 `redis.conf` 配置文件中所有内存相关的指令。`redis.conf` 文件为大多数指令提供了丰富的内联文档，使得一些复杂的内存优化选项易于理解、更改和测试。大多数 Redis 配置指令可以在运行时

通过 `CONFIG SET` 命令进行设置。

本章中 LRU 键驱逐主题中包含了 LRU 相关的配置指令。

首先，我们要讨论的配置指令是 `rdbchecksum` 指令，它的默认值为 `yes`，将 65 位循环冗余检验码（CRC64）放置在 RDB 快照文件的末尾，作为防止文件损坏的一种手段。当 Redis 产生一个子进程将快照保存至磁盘时，对 RDB 快照进行 CRC64 校验会增加 10% 的内存使用。

接下来，我们要研究的是 `activeremhashing` 配置指令。Redis 中的主哈希表将主键与对应的值进行关联。如果将 `activeremhashing` 指令设置为 `yes`，那么该主哈希表每隔 100 毫秒会重新哈希。重新哈希的过程将释放删除了的键占用的操作系统内存。由于 `activeremhashing` 发生在非工作时间，因此对客户端连接的影响最小。就像 `redis.conf` 的注释中推荐的那样，如果有硬性的延迟需求，应当把 `activeremhashing` 设置为 `no`。否则，如果 Redis 服务器需要支持高并发量客户端，可能会在运行重新哈希期间造成延迟。

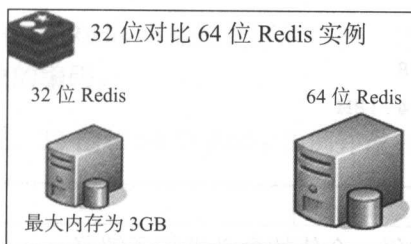
主从复制

Redis 的一个激动人心的功能是可伸缩性，它采用主从复制的方式提供高度可靠性。在集群环境中，我们可以通过 `slave-of` 指令将 Redis 实例切换到从模式，此时从实例被允许从另一个被指派为主实例的 Redis 中复制数据。内存和来自硬件和网络的延迟会直接影响主从实例的性能。在主从模式下提升 Redis 的冗余度是在内存、硬件和网络流量之间做出的权衡。

`repl-disable-tcp-nodelay` 指令可以用来更好地处理 Redis 主从实例间的网络流量拥塞。通过对主从间的密集数据同步和更少的网络流量之间进行权衡，这种复制能够改善高流量情况下的网络性能。

32 位 Redis

在 Redis.io 网站上的官方文档内存优化中，其中一条建议是在 32 位模式下编译 Redis 替代默认的 64 位实例。



对于同样小于 3GB 的数据集,其在 32 位 Redis 实例中要比在 64 位版本的 Redis 中小。下列测试展示了这一结果。我们将启动两个 Redis 实例,INSTANCE64 和 INSTANCE32。我们将在 Python 命令行上创建一个快速 Python 函数 test_redis_32k_65k,使用 UUID 作为字符串值创建 100,000 个键:

```
>>>def test_redis_32k_64k():
    for i in range(100000):
        key = "uuid:{}".format(i)
        value = uuid.uuid4()
        INSTANCE32.set(key, value)
        INSTANCE64.set(key, value)
>>> test_redis_32k_64k()
```

为了能清楚地了解 32 位和 64 位 Redis 实例的内存使用状况,我们将分别连接这两个实例,比较 redis-cli 会话的输出。

以下是 Redis 32 位实例的输出:

```
127.0.0.1:6378> INFO memory
```

```
# Memory
```

```
used_memory:12447072
```

```
used_memory_human:11.87M
```

```
used_memory_rss:13733888
```

```
used_memory_peak:12447072
```

```
used_memory_peak_human:11.87M
```

以下是 Redis 64 位实例的输出:

```
127.0.0.1:6379> INFO memory
```

```
# Memory
```

```
used_memory:14871888
```

```
used_memory_human:14.18M
```

```
used_memory_rss:16805888
used_memory_peak:14871888
used_memory_peak_human:14.18M
```

INFO memory 详解

以下对 INFO memory 的每一个值的含义进行了解释：

- `used_memory`: 通过 `libc`、`jemalloc` 或者 Redis 使用的其他分配方法分配的字节数大小
- `used_memory_human`: 将 `used_memory` 格式化为人类可读的值，以 MB 为单位
- `used_memory_rss`: 常驻集大小（**Resident set size**，简写为 **rss**）指的是在操作系统中看到的内存分配，以及通过 UNIX 工具 `top` 显示的结果
- `used_memory_peak`: Redis 使用的峰值内存，以字节为单位
- `used_memory_peak_human`: 将 `used_memory_peak` 格式化为人类可读的格式，以 MB 为单位
- `used_memory_lua`: Redis 的 Lua 子系统使用的字节数
- `mem_fragmentation_ratio`: `used_memory_rss` 与 `used_memory` 的比率
- `mem_allocator`: 在编译期 Redis 使用的分配器

我们看到这个简单的例子中使用完全相同的数据，32 位 Redis 实例使用了 11.87MB 的内存，而 64 位 Redis 实例使用了 14.18MB 的内存。由于采用了较小的 100,000 个键的样本，并且采用了 Redis 字符串数据结构，导致两者之间相对较小的差异。为了有助于展示采用 32 位版本对比 64 位 Redis 版本需要考虑的取舍及可能遇到的问题，我们将比较一百万的整数、浮点数、Redis 字符串、列表、哈希，以及集合数据结构，并将结果列在以下表格中。

	32 位峰值	64 位峰值	差异
通过 SET 命令使用整数 1 来设置 1,000,000 个键	35.19 MB	54.79 MB	64 位多使用了 19.6MB（即 36%）
通过 SET 命令使用浮点数 3.142 来设置 1,000,000 个键	65.71 MB	85.26 MB	64 位多使用了 19.55MB（即 22%）
通过 SET 命令使用 385e7bc8-0075-4922-9dfa-a0b2592d5c78 字符串来设置 1,000,000 个键	96.23 MB	115.77 MB	64 位多使用了 19.54MB（即 16%）

如果应用程序使用整数集合，只要总内存不超过 4GB 的最大限制，那么 32 位版本节省的内存是相当可观的。当存入不同大小和类型的值时，Redis 的 32 位和 64 位版本之间的百分比差异意味着节省的内存数量显著降低。对那些使用集合的应用程序来说，它们较多

地使用了字符串，因而 64 位的 Redis 可能是更好的选择。这是因为在 64 位版本中的字符串拥有额外的空间及更高效的编码。

对于哈希数据结构来说，32 位和 64 位 Redis 实例并没有特别大的差异，如下表所示：

	32 位峰值	64 位峰值	差异
添加 1,000,000 个值为 1 的属性到 Redis 哈希中	50.15 MB	69.69 MB	64 位多使用了 19.54MB (即 28%)
添加 1,000,000 个值为 3.142 的属性到 Redis 哈希中	80.68 MB	100.24 MB	64 位多使用了 19.56MB (即 19%)
添加 1,000,000 个值为 385e7bc8-0075-4922-9dfa-a0b2592d5c78 的属性到 Redis 哈希中	111.18 MB	130.74 MB	64 位多使用了 19.56MB (即 14.9%)

对于 Redis 哈希来讲，无论存储了什么类型的值，32 位和 64 位的开销基本定格在了 19.56。

对 Redis 列表采取同样的测试，我们得到了以下峰值内存结果：

	32 位峰值内存	64 位峰值内存	差异
添加 1,000,000 个连续的整数到 Redis 列表中	46.15 MB	61.69 MB	64 位多使用了 15.54MB (即 25%)
添加 1,000,000 个连续的浮点数到 Redis 列表中	46.45 MB	77.24 MB	64 位多使用了 30.78MB (即 39.8%)
添加 1,000,000 个字符串 385e7bc8-0075-4922-9dfa-a0b2592d5c78 到 Redis 列表中	76.97 MB	92.51 MB	64 位多使用了 15.54MB (即 16.8%)

对于 32 位版本的 Redis 列表来说，在 32 位限制之下，列表非常适合存储整数和浮点数。基于字符串的测试则显示出 32 位和 64 位之间内存使用效率提升最小。

接下来，我们将运行 3 个采用 Redis 集合的测试，用于比较 32 位和 64 位下的性能：

	32 位峰值内存	64 位峰值内存	差异
将 1,000,000 个整数添加到 Redis 集合中	50.15 MB	69.71 MB	64 位多使用了 19.56MB (即 28%)
将 1,000,000 个浮点数添加到 Redis 集合中	50.45 MB	85.24 MB	64 位多使用了 34.79MB (即 40.8%)
将 1,000,000 个唯一的 UUID 字符串添加到 Redis 集合中	80.97 MB	108.53 MB	64 位多使用了 27.56MB (即 25.39%)

以下是一些关于使用 32 位版本 Redis 的注意事项。32 位的 Redis 并未在 Redis 用户群

中进行广泛部署和测试，因此相较于 64 位版本可能会有未发现的 BUG。另一个需要注意的地方是，诸如 BITOP 和 BITCOUNT 这样的位操作基于 Redis 的 64 位版本进行了优化，因此相对而言 32 位则没有这么高效。最后，在 32 位 Redis 中设置 maxmemory 参数更困难（我们会针对缓存进行更深入地研究），这是因为如果在 32 位版本的 Redis 中的 maxmemory 值设置得过于靠近最大值 4GB，那么通信、主从复制、I/O 缓存都有可能随时导致 Redis 崩溃。

键过期

保证 Redis 数据库不会超过可用内存的简单又可靠的方法是为键设置超时时间，一旦过了键的超时时限，键就会被自动驱逐。如果应用程序无须存储过时陈旧的数据，为键空间设计一套高效的过期策略将使得应用程序的内存需求更可控。一种流行的使用键过期策略的 Redis 设计模式是将过期的或者被驱逐的数据保存到另一个关系型 SQL 数据库或者其他基于磁盘的 NoSQL 平台，例如 MongoDB。

在应用中实现键过期功能时需要注意以下几个方面。首先，当在键上调用 EXPIRE 命令设置过期时间时，该超时只能通过删除或者替换键的方式清除。之后，任何改变值的命令是无法更改或者清除之前设置的超时的。让我们假设这样的场景，你正在编写有关冲泡三类茶的应用。每种类型的茶拥有不同的冲泡时间。

我们的 Redis 键模式将选用正斜杠代替冒号作为分隔符，如果之后我们想添加 REST 服务，这样方便记忆。同时，我们将使用递增的唯一计数器。每一个茶盒都是整形集合，用于存放茶袋的总数。最后，当应用程序开始冲泡茶袋时，我们将从茶盒里取出随机的整数，用于键计数器，并以推荐的冲泡时间长度设置过期时间。当茶袋键被驱逐时，应用程序就停止冲泡。现在我们可以建模并测试上述用例场景。示例中的 Python 函数中，接收 Python 客户端实例、名称、冲泡时间和茶盒大小作为参数，并返回最新创建的 tea_key 以便应用后续使用。

```
def create_tea(datastore, name, time, size):
    # Increment and save global counter
    tea_counter = datastore.incr("global/teas")
    tea_key = "tea/{}".format(tea_counter)
    datastore.hmset(tea_key,
        {"name": name,
         "brew-time": time,
         "box-size": size})
```

```
return tea_key
```

现在，我们引入 Python 的 Redis 模块，实例化 StrictRedis 类，并使用以下函数从 Python shell 创建三类茶并存储为哈希类型。

```
>>> import redis
>>> tea_datastore = redis.StrictRedis()
>>> earl_grey = create_tea(tea_datastore, "Earl Grey", 5, 15)
>>> earl_grey
'tea/1'
>>> tea_datastore.hgetall(earl_grey)
{'box-size': b'15', b'name': b'Earl Grey', b'brew-time': b'5'}
>>> lavender_mint = create_tea(tea_datastore, "Lavender Mint", 2,
20)
>>> peppermint_punch = create_tea(tea_datastore, "Peppermint Punch",
4, 10)"""
```

为每种类型的茶袋单独添加到各自的第一个茶盒中，然后创建第二个函数：

```
def add_box_of_tea(datastore, tea_key, number):
    box_counter = datastore.incr("global/{}/boxes".format(tea_key))
    tea_box_key = "{} /box/{}".format(tea_key, box_counter)
    datastore.sadd(tea_box_key, *range(1, number+1))
    return tea_box_key
```

我们从 Python shell 上为每种类型的茶添加第一个茶盒，如下所示：

```
>>> earl_grey_box_1 = add_box_of_tea(tea_datastore, earl_grey, 15)
>>> earl_grey_box_1
'tea/1/box/1'
>>> tea_datastore.smembers(earl_grey_box_1)
{'b'2', b'1', b'3', b'5', b'4', b'13', b'10', b'11', b'14', b'7',
b'12', b'15', b'6', b'8', b'9'}
>>> lavender_mint_box_1 = add_box_of_tea(tea_datastore,
lavender_mint, 15)
>>> tea_datastore.scard(lavender_mint_box_1)
15
>>> peppermint_punch_box_1 = add_box_of_tea(tea_datastore,
peppermint_punch, 10)
```

```
>>>tea_datastore.scard(peppermint_punch_box_1)
10
```

现在，我们将添加第三个函数，它将 Redis 实例和茶盒键作为参数，从每种茶盒中随机抽取一包进行冲沏，并将过期时间设置为此类型茶的冲沏时间，将茶袋键添加到冲沏集合中，最后返回该 tea_bag_key:

```
def start_brew(datastore, tea_box_key):
    tea_box = tea_box_key.split("/box")[0]
    # Brew time is in minutes, we multiple by 60 for expire in
    seconds
    # 冲沏时间是以分钟为单位的，因此这里需要乘以60换算成秒
    expire_time = int(datastore.hget(tea_box, "brew-time"))*60
    tea_bag_number = datastore.spop(tea_box_key)
    tea_bag_key = "{} /bag/{}".format(tea_box_key,
        tea_bag_number.decode())
    datastore.set(tea_bag_key, "brew")
    datastore.expire(tea_bag_key, expire_time)
    datastore.sadd("brewing", tea_bag_key)
    return tea_bag_key""""""
```

分别为每种类型的茶调用 start_brew 函数，一共三次，创建三个茶袋键并依据茶类型设置过期时间。

最后一个 Python 函数遍历 brewing 集合中所有的茶袋，通过 TTL 命令查询每个茶袋的剩余时间。同时根据茶叶数据库中该茶袋键过期前剩余多少时间，要么在茶叶完成冲沏之前打印剩余时间的消息，要么打印该茶叶已经冲沏完毕可以饮用了：

```
def poll_brewing(datastore):
    active_tea_bags = datastore.smembers("brewing")
    for tea_bag in active_tea_bags:
        time_left = datastore.ttl(tea_bag)
        if time_left > 0:
            print("{} seconds left for {}".format(time_left,
                tea_bag))
        else:
            print("{} Ready to Drink!".format(tea_bag))
            # Remove expired tea bag from brewing set
```

```
datastore.srem("brewing", tea_bag)

peppermint_punch, 10)
>>>tea_datastore.scard(peppermint_punch_box_1)
```

我们一开始就调用 3 次 poll_brewing Python 函数:

```
>>>poll_brewing(tea_datastore)
80 seconds left for b'tea/2/box/1/bag/5'
215 seconds left for b'tea/3/box/1/bag/3'
243 seconds left for b'tea/1/box/1/bag/5'
```

大约 60 秒后, 再次调用:

```
>>>poll_brewing(tea_datastore)
22 seconds left for b'tea/2/box/1/bag/5'
157 seconds left for b'tea/3/box/1/bag/3'
185 seconds left for b'tea/1/box/1/bag/5'
```

最后在 90 秒的时候:

```
>>>poll_brewing(tea_datastore)
b'tea/2/box/1/bag/5' Ready to Drink!
124 seconds left for b'tea/3/box/1/bag/3'
152 seconds left for b'tea/1/box/1/bag/5'
```

如果设置了超时时间的键的值被修改, 例如向 Redis 字符串使用 APPEND 命令, 该超时设置仍然会继续。通过 redis-cli 会话, 我们为一个茶袋设置字符串值和 300 秒过期时间:

```
127.0.0.1:6379> SET tea/1/box1/bag/8 brew
OK
127.0.0.1:6379> EXPIRE tea/1/box1/bag/8 300
(integer) 1
```

首先, 我们检查键 tea/1/box/1/bag/8 的 TTL 值, 然后为该键的值添加额外的文本, 并再次检测 TTL 值, 结果如下:

```
127.0.0.1:6379> TTL tea/1/box1/bag/8
(integer) 288
127.0.0.1:6379> APPEND tea/1/box1/bag/8 ing
(integer) 7
```



```
127.0.0.1:6379> GET tea/1/box1/bag/8
"brewing"
127.0.0.1:6379> TTL tea/1/box1/bag/8
(integer) 259
```

如果在设置了超时的键上调用 SET 或者 GETSET 命令，那么超时会被清除，键不会从数据库中驱逐。因此，如果在 tea/3/box1/bag/3 key 过期前调用了 SET 命令，那么调用该键的 TTL 命令时 Redis 将返回-1。这是在没有设置超时时默认返回的消息。

```
127.0.0.1:6379> TTL tea/3/box1/bag/3
(integer) 225
127.0.0.1:6379> SET tea/3/box1/bag/3 brew
OK
127.0.0.1:6379> TTL tea/3/box1/bag/3
(integer) -1
```

到目前为止，我们将 TTL 作为拉取机制获取我们所关心的剩余时间。使用客户端拉的模式有其不足之处，其中之一就是客户端代码和服务端之间会产生延迟，导致我们的茶可能冲沏过头。Redis 提供了一种基于发布/订阅模式的通知机制，我们会在之后的章节中介绍 Redis 的消息通信，在这里我们可以通过该机制设置当键过期时发送消息。

你也能使用 PERSIST 命令清除键上的超时设定。最后，在一个已经设置过超时的键上调用 EXPIRE 命令将会清除并重新设定超时。

```
127.0.0.1:6379> TTL tea/1/box1/bag/8
(integer) 118
127.0.0.1:6379> EXPIRE tea/1/box1/bag/8 300
(integer) 1
127.0.0.1:6379> TTL tea/1/box1/bag/8
(integer) 295
```

即便上述实例是刻意杜撰，它也展示了 Redis 键过期的基本操作。Redis 通过 EXPIRE 命令使用当前运行的操作系统的绝对 UNIX 时间戳来设置 TTL。如果为一个键设置了超时，但是此时关闭 Redis 数据库，将数据存储到快照中的话，在超时之后重启 Redis 数据库会自动驱逐该键。Redis 使用两种方法处理键空间中的过期操作。第一种方法是该键是否正被客户端请求，第二种方法是一种概率算法，使用相关的过期时间戳随机测试 20 个键，并删除结果中所有已过期的键。

在本章的下一个主题中，我们将带着键过期的知识，来看看当达到最大内存限制并且键空间中的键设置了超时的时候，Redis 是如何进行自我调整的。

LRU 键驱逐策略

为了演示 Redis 中不同的键驱逐选项，我们将从一个简单的示例开始，通过 `maxmemory` 指令将最大内存设置为 1MB 以创建一个小内存的 Redis 实例。`maxmemory` 指令允许设定内存大小的硬性上限，运行时的 Redis 实例受限于此。就像默认的 `redis.conf` 文件中警告的那样，我们将看到设置 `maxmemory` 所带来的后果。为了便于开始，我们创建了一个非常简单的 Redis 键模式，用于为 Web 应用程序产生并存储唯一 id。在通过 `redis-cli` 连接到 Redis 实例之后，我们将运行以下命令来清除我们的数据库，之后将 `maxmemory` 指令设置为 1MB。

```
127.0.0.1:6379> FLUSHALL
OK
127.0.0.1:6379> CONFIG SET maxmemory1mb
OK
```

下一步，我们将实现一个函数，接收 Redis 实例作为参数，自增全局 `uuid`，并使用标准 `uuid` Python 模块产生随机的 UUID。下面这部分代码片段是用 Python 编写的 `add_id` 函数代码：

```
>>> import uuid
>>> def add_id(redis_instance):
redis_key = "uuid:{}".format(redis_instance.incr("global:uuid"))
redis_instance.set(redis_key, uuid.uuid4())
```

当 Redis 内存耗尽时，默认生效的是永不过期策略（`noeviction policy`），如下图所示：

noeviction 策略

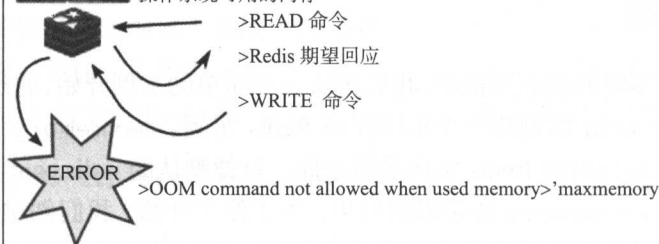
问题：当内存耗尽时，Redis 数据库会发生什么？

noeviction

Redis 的默认策略，当没有可用内存时，如果 Redis 尝试写入数据库就会返回错误。

max-memory 该配置指令用于限制内存

操作系统可用的内存



默认的 **maxmemory-policy** 策略是永不过期。在 **noeviction** 策略中，没有键设置为过期。如果 Redis 没有可用内存，任何写操作都会导致 Redis 错误。为了确保指令被正确设置，我们将首先检查是否设置成了 1MB 及是否设置成了 **noeviction** 策略。检查的方式是在 **redis-cli** 会话中运行 **CONFIG GET** 命令来检测返回的值。

```
127.0.0.1:6379>CONFIG GET maxmemory
1) "maxmemory"
2) "1048576"
127.0.0.1:6379>CONFIG GET maxmemory-policy
1) "maxmemory-policy"
2) "noeviction"
```

为了测试 **noeviction** 策略，我们将一直循环该 Python 脚本，直到收到错误提示为止，如下所示：

```
>>> while 1:
add_id(local_redis)
```

很快，我们就接收到来自 Redis 客户端的错误提示，如下所示：**redis.exceptions.ResponseError**，**OOM command not allowed when used memory > 'maxmemory'**。

在到达 1MB 内存上限前，**while** 循环一共执行了 181 次，循环指针定位在了 181。现在，让我们通过 **redis-cli** 执行 **INFO memory** 命令，查看我们的数据库状态，特别留意 **used_memory_peak_human**：

```
127.0.0.1:6379> INFO memory
# Memory
```

```

used_memory:1048608
used_memory_human:1.00M
used_memory_rss:1769472
used_memory_peak:1048608
used_memory_peak_human:1.00M
used_memory_lua:35840
mem_fragmentation_ratio:1.69
mem_allocator:libc

```

现在, 如果我们获取在 `add_id` 函数中设置的全局 `uuid` 计数器键, 我们发现在 1MB 数据存储中存储的 181 UUID 和我们在循环中递增的计数器变量是一致的:

```

127.0.0.1:6379> GET global:uuid"181"
127.0.0.1:6379> GET uuid:181
"1930a94e-38ff-4dbd-8885-eb44aed96122"

```

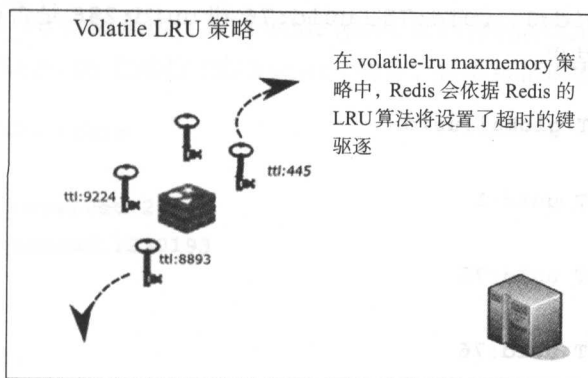
我们将从 `redis-cli` 上通过尝试自增另一个变量的方式测试 `noeviction` 策略, 例如 `tmp:1`, 我们收到的错误提示如下:

```

127.0.0.1:6379> INCR tmp:1
(error) OOM command not allowed when used memory > 'maxmemory'.

```

当 Redis 在无可用内存的情况下, 尝试执行任何写操作 (`SET`、`INCR`、`SADD`、`HSET` 等) 或者其他会增加内存使用的命令时, 都会收到类似于前面代码片段那样的错误提示。下一个我们要研究的策略是当一个或多个键设置了超时的时候, Redis 是如何处理 LRU 的。首先第一种过期 LRU 策略名为 `volatile-lru`, 它将最近较少使用的键驱逐出去。这些键必须是通过 `EXPIRE SET` 命令设置了超时的。如果没有键符合条件而被驱逐出内存的话, Redis 会像 `noeviction` 策略那样, 在写入命令时返回同样的异常信息。



使用该策略时需要特别注意的一点是，当 Redis 内存耗尽时，Redis 会开始删除那些设置了过期时间的键，即便该键仍然有剩余时间。为了测试 `volatile-lru` 策略，我们将清空 Redis 实例。注意，如果不为任何键设置驱逐时间，那么运行同样的循环将导致和 `noeviction` 策略一样的结果：

```
127.0.0.1:6379>FLUSHDB
127.0.0.1:6379>CONFIG SET maxmemory-policy volatile-lru
127.0.0.1:6379> GET global:uuid
"181"
127.0.0.1:6379> INFO memory
# Memory
used_memory:1048608
used_memory_human:1.00M
.
.
```

现在，我们将基于 `add_id` 创建第二个函数，并使用新的 `add_id_expire` 函数为我们创建的前 75 个键设置 300 秒超时时间。

```
>>>def add_id_expire(redis_instance):
    count = redis_instance.incr("global:uuid")
    redis_key = "uuid:{}".format(count)
    redis_instance.set(redis_key, uuid.uuid4())
    if count <= 75:
        redis_instance.expire(redis_key, 300)
```

将计数器变量重设为 0，并再次运行测试程序，此时我们一共循环了 238 次，相较使用 `noeviction` 策略运行 Redis 实例时多出了 57 次。当我们获取全局递增的 `global:uuid` 变量值，分别检测 `uuid:1`、`uuid:75`、`uuid:76` 和 `uuid:238` 是否存在于数据库中，就可以确认程序循环的结果：

```
127.0.0.1:6379> GET global:uuid
"238"
127.0.0.1:6379> GET uuid:1
(nil)
127.0.0.1:6379> GET uuid:75
(nil)
127.0.0.1:6379> GET uuid:76
```

```
"9922e314-17f8-4630-a709-07a3c8a8019c"
```

```
127.0.0.1:6379> GET uuid:238
```

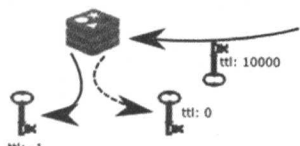
```
"1a1318ae-57b6-4a4a-a366-2727033a315d"
```

如同我们所期望的那样,相较于 noeviction 策略,从 uuid:1 到 uuid:75,这些键都不复存在了。同时,在这次循环中,数据库中创建了额外的 57 个键。我们也可以发现我们没有创建额外的 75 个 uuid 而是 57 个。这很有可能是在尝试添加更多的键而没有考虑键驱逐需要的内存开销,导致在所有设置了过期时间的键被驱逐出内存之前耗尽了内存。

实战 allkeys-lru

Redis 中的 allkeys-lru 策略基于键的 TTL (键到期剩余时间) 来驱逐键

Redis 的键值存储到数据库时,同时使用 EXPIRE 命令设置超时时间



当有新的键写入 Redis,同时 Redis 实例已经到达/或者接近于最大内存限制时,Redis 会将设置了 TTL 的键依据 LRU 算法进行驱逐,而忽略那些没有设置超时时间的键。

现在,下一个 LRU 风格的驱逐策略是 allkeys-lru。如果你期望为 Redis 数据库添加幂律访问模式,那么推荐使用该驱逐策略。重要的是,当你对选用哪种驱逐策略举棋不定时,allkeys-lru policy 是一个很好的初始选择。allkeys-lru 策略会删除 Redis 中任何一个键,而且没有办法限制哪些键被删除。如果应用程序需要持久化部分 Redis 键(例如为了配置或者引用查找),那么千万不要使用 allkeys-lru 策略!

为了测试 allkeys-lru,我们将 Redis 实例中的数据清除,设置好 maxmemory-policy 指令,然后运行最初的 add_id 函数。

运行最初的 add_id 函数来执行无限 while 循环,在内存耗尽之前,共执行了几十万次迭代 (615,094)。在 Redis-cli 上运行 INFO stats 命令,查看被清除键的总数,如下所示:

```
127.0.0.1:6379> INFO stats
```

```
# Stats
```

```
total_connections_received:2
```

```
total_commands_processed:1230193
```

```
.
```

```
.
```

```
evicted_keys:264524
```

在策略中, Redis 可以处理 1,230,193 个命令, 驱逐 264,524 个键。由于循环中的所有键都有着同样的用法 (也就是说, 我们并没有在最初 SET 命令之后获取过任何一个键), 因此 Redis 对数据存储中的所有键的 LRU 评判标准是一致的。就我们的实验来说, allkeys-lru eviction 策略在通过驱逐数据库中陈旧的键, 从而为额外的键释放内存方面是非常高效的。为了进一步测试 allkeys-lru 策略, 我们重启实验, 但这次仅循环 200 次 (多余针对 noeviction 策略测试的量, 但少于针对 volatile-lru 测试的量)。本次运行结果如下:

```
127.0.0.1:6379> GET global:uuid
"200"
127.0.0.1:6379> INFO stats
# Stats
total_connections_received:2
total_commands_processed:614
.
.
evicted_keys:24
```

在本次测试中, allkeys-lru 策略驱逐了 24 个键, 我们创建了完整的 200 个键是因为我们期望程序走完 200 次循环, 但问题在于我们不清楚 200 个键中哪些被驱逐了。为了确定缺失了哪些键, 我们需要循环所有的键, 并测试每一个键是否存在还是已经被驱逐了。可以通过以下 Python 代码片段轻松完成:

```
>>> for i in range(1, 201):
    key = "uuid:{}".format(i)
    if not local_redis.exists(key):
        print(key)
```

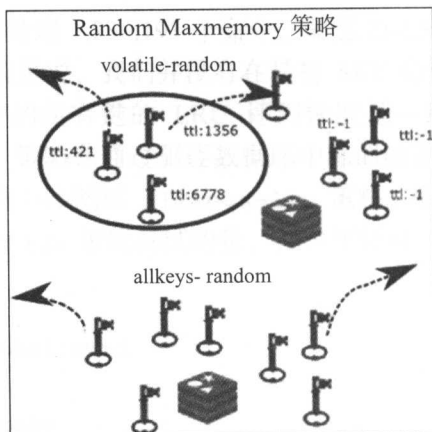
以下是该代码片段的输出 (在你机器上运行这段代码得到的结果应该和输出的列表有所不同, 如果只是些许不同, 很可能是因为 Redis 的 LRU 算法的概率本质使然):

```
uuid:15
uuid:17
uuid:23
uuid:29
uuid:39
uuid:46
uuid:50
```

```
uuid:57  
uuid:67  
uuid:68  
uuid:83  
uuid:86  
uuid:89  
uuid:110  
uuid:116  
uuid:121  
uuid:128  
uuid:130  
uuid:146  
uuid:147  
uuid:150  
uuid:151  
uuid:175  
uuid:176
```

针对这些被驱逐的键展示了 Redis 的 LRU 算法, 其中有点需要注意的地方。首先, Redis 的 LRU 算法是不准确的, 因为 Redis 并不会自动选择最佳的候选键来驱逐, 例如最少使用的键或者是最早访问的键。相反, Redis 默认的行为是选取 5 个键的样本, 并驱逐当中最少使用的那个。回顾之前被驱逐了的键的列表, 我们可以从中了解到这种取样策略所呈现的结果。如果我们想要增加 LRU 算法的精确性, 我们可以通过更改 `redis.conf` 文件中的 `maxmemory-samples` 指令, 或者在运行时通过 `CONFIG SET maxmemory-samples` 命令进行设置。将样本大小增加到 10, 从而提升 RedisLRU 算法的性能, 效果接近真实 LRU 算法, 但是副作用就是消耗更多的 CPU 计算能力。

将样本大小降至 3, 从而减少了 RedisLRU 算法的精确性, 不过相应地加快了处理速度。



接下来是两种最大内存驱逐策略：volatile-random 和 allkeys-random。它们与 volatile-lru 和 allkeys-lru 相似，但是不采用 LRU 算法。volatile-random 策略基于键上设置的过期状态随机驱逐一个键。对于 allkeys-random 策略来说，整个键空间都是开放的。

我们可以使用 add_id 和 add_id_expire 函数测试这两种策略。首先，我们使用修改过的 add_id_expire 函数对 volatile-random 策略进行实验。该函数会将半数的键设置 5 分钟的过期时间。这样的设定允许我们将该策略同其他策略在性能上进行比较。我们总共创建了 246 个键，实验结果与 volatile-lru 测试的结果不尽相同。不同于 volatile-lru 策略，我们需要 $O(n)$ 时间复杂度的操作来计算我们创建的这些键是否已被驱逐。

运行 1000 次 add_id_expire 函数将产生如下性能数据：

```
127.0.0.1:6379> INFO stats
# Stats
total_connections_received:2
total_commands_processed:1805
.
.
expired_keys:0
evicted_keys:499
```

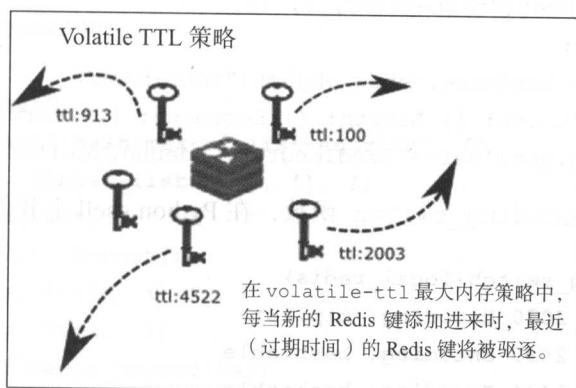
注意，即便 add_id_expire 函数为添加到 Redis 实例中的半数键设置了过期时间，在 volatile-lru 策略下，样例数据集中的一小部分在键过期之前全部被驱逐出了内存。通过检查键空间的状态，我们可以看到如下所示：

```

127.0.0.1:6379> GET global:uuid
"652"
127.0.0.1:6379> GET uuid:1
(nil)
127.0.0.1:6379> DBSIZE
(integer) 153
127.0.0.1:6379> GET uuid:652
(nil)
127.0.0.1:6379> GET uuid:651
"e42ce917-efe9-4657-b2d6-cccd0f26b19c"

```

在对 volatile-random 策略的实验中，还没有完成 1000 次迭代我们就耗尽了内存。通过调用 GET global:uuid call 命令确认进行了 652 次迭代，我们创建并驱逐了 499 个键，同时只保留了 153 个键。



最后一个最大内存策略是 volatile-ttl。它与 volatile-lru 相似，不过有着额外的特性。Redis 会尝试根据键的剩余时间（TTL）清除键。

创建内存高效的 Redis 数据结构

接下来是有关 Redis 内存优化的方法：

小巧的哈希、列表、集合和有序集合

对于哈希、列表和有序集合来说，这种特殊编码方式基于 ziplist（压缩列表）。在

ziplist.c 文件中针对压缩列表有以下描述：

压缩列表是一种特殊的双向链表，专为内存高效进行设计。它存储了字符串和整数值，其中整数是以真正的整数值而非字符串形式进行存储的。它允许在列表的任意一端以 $O(1)$ 时间复杂度进行 push 和 pop 操作。但是，由于每次操作都需要为 ziplist 使用的内存进行重新分配，实际上的复杂度与 ziplist 所使用的内存大小有关。

根据大小、类型及数据结构的内容，ziplist 编码方式为 Redis 数据库极大地节约了内存使用。Redis 会依据 Redis 数据类型的限制，动态地在 ziplist 和默认的数据结构编码两者之间进行切换。为了在实战中观察这种切换，我们将创建 Python 函数，通过打印编码类型及哈希键的大小展示当到达默认限定时的这种动态切换：

```
def dynamic_encoding_switch(instance):
    for i in range(515):
        instance.hset("test-hash", i, 1)
        if i > 510:
            debug = instance.debug_object("test-hash")
            print("Count: {} Length: {} Encoding: {}".format(i,
                debug.get('serializedlength'), debug.get('encoding')))
```

运行 dynamic_encoding_switch 函数，在 Python shell 上我们可以看到如下结果：

```
>>>dynamic_encoding_switch(local_redis)
Count: 511 Length: 2070 Encoding: ziplist
Count: 512 Length: 2439 Encoding: hashtable
Count: 513 Length: 2444 Encoding: hashtable
Count: 514 Length: 2449 Encoding: hashtable
```

在该实例中，Redis 为何要动态地为哈希表从 ziplist 重新编码为哈希表呢？这是出于对内存效率和性能两者的权衡。Redis 中 ziplist 的实现通过为每个节点只存储 3 份数据实现较小的内存占用；第一份是前一个节点的长度，第二份是当前节点的长度，第三份是存储的值。

对于哈希表来说，hash-max-ziplist-entries 指令设置了总共有多少字段时可以被特殊编码为 ziplist，默认值是 512。hash-max-ziplist-value 指令设置了从 ziplist 转变为哈希表所要达到的最大大小，默认是 64。我们可以通过接下来一个非常简单的例子展示这两个条件的作用。

为了测试对哈希来说压缩列表和链表间的大小差异，让我们启动两个完全一样的 Redis

实例。为了本章之后的测试，我们将保存第一个实例的配置文件指令，使用的是 Redis 默认值，同时修改第二个实例的配置，强制使用每个数据类型的默认编码。

警告



需要注意的是，如果你正尝试通过为已存在的数据库调整集合、哈希和列表的 ziplist 阈值提升内存性能，任何之前已经存在的值将保持原来的编码格式。更改这些阈值不会对之前的数据生效，只对新添加到 Redis 的值有效。

首先，我们将第二个实例的 `hash-max-ziplist-entries` 值设置为 0，创建一个哈希并填充 500 个完全相同的字段和整数值，然后从 `redis-cli` 上使用 `DEBUG OBJECT` 命令对两者进行比较。首先，我们将新建 Python 函数来创建哈希：

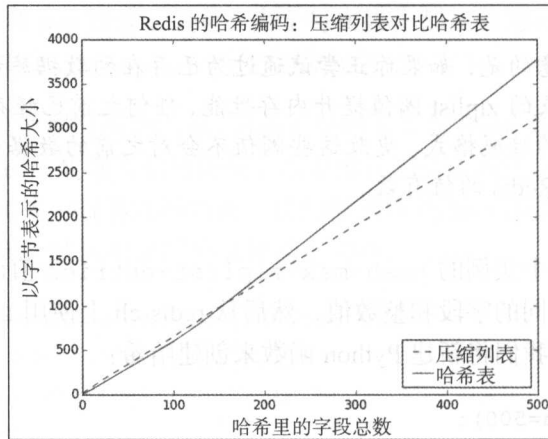
```
def plot_hashes(runs=500):
    reset()
    key = "test-hash"
    INSTANCE2.config_set('hash-max-ziplist-entries', 0)
    run, zip_list, linked_list = [], [], []
    for i in range(runs):
        field = "f{}".format(i)
    INSTANCE1.hset(key, field, i)
    INSTANCE2.hset(key, field, i)
    debug1 = INSTANCE1.debug_object(key)
    zip_list.append(debug1.get("serializedlength"))
    debug2 = INSTANCE2.debug_object(key)
    linked_list.append(debug2.get("serializedlength"))
```

现在，我们运行该函数，然后在 `test-hash` 上为每个对象检查 `debug_object` 命令返回的结果，如下所示：

```
>>>plot_hashes()
>>INSTANCE1.debug_object("test-hash").get("serializedlength")
3102
>>>INSTANCE2.debug_object("test-hash").get("serializedlength")
3764
```

如果比较这两个完全一样的哈希的编码大小，会发现 `test-hash` 的标准哈希表编码的序

列化长度为 3764，而 test-hash 的压缩列表长度为 3102，可以直观地看到节省了 662 字节的内存（在序列化的情况下）。将结果绘制成下图，你只能从中看到节省的内存，但是该图没有考虑到随着哈希大小的增长，压缩列表编码需要的额外计算时间：



对列表来说，像哈希一样，ziplist 编码用于小型列表，阈值是由 list-max-ziplist-entries 和 list-max-ziplist-value 这两个指令决定的，它们和哈希一样默认值分别为 512 和 64。在代码文件 small_types_tests.py 中，针对第二个 Redis 实例将 list-max-ziplist-value 设置成了 0。然后该函数迭代 runs 值，为每个 Redis 实例中相同的 Redis 键设置随机 UUID 值，并将每个实例的序列化长度保存起来。以下是 Python 代码片段 small_types_tests.py：

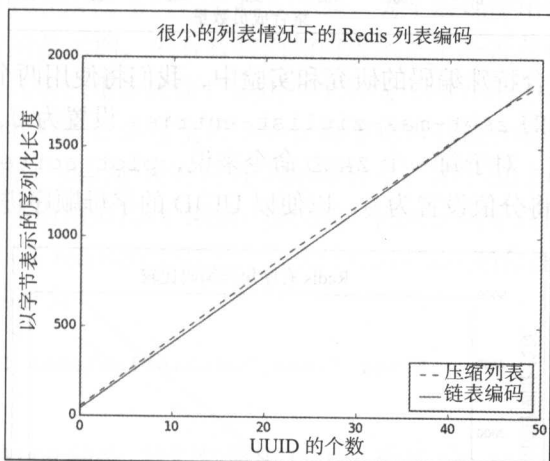
```
def plot_list_ziplist(runs=1000):
    reset()
    INSTANCE2.config_set("list-max-ziplist-entries", 0)
    key = "test-list"
    run, zip_list, linked_list = [], [], []
    for i in range(runs):
        run.append(i)
        uid = uuid.uuid4()
        INSTANCE1.lpush(key, uid)
        INSTANCE2.lpush(key, uid)
        debug1 = INSTANCE1.debug_object(key)
        zip_list.append(debug1.get("serializedlength"))
        debug2 = INSTANCE2.debug_object(key)
        linked_list.append(debug2.get("serializedlength"))
```

我们可以通过比较两个 Redis 实例中的 test-list 的两个列表编码方法观察差异：

```
>>>INSTANCE1.debug_object("test-list").get("serializedlength")
15756
>>>INSTANCE2.debug_object("test-list").get("serializedlength")
18946
```

对于第一批添加到两个列表中的 512 个 UUID 来说，压缩列表编码的大小为 15,787，而链表编码大小为 18,946，也就是说，如果使用压缩列表能节省 3190 字节的内存。

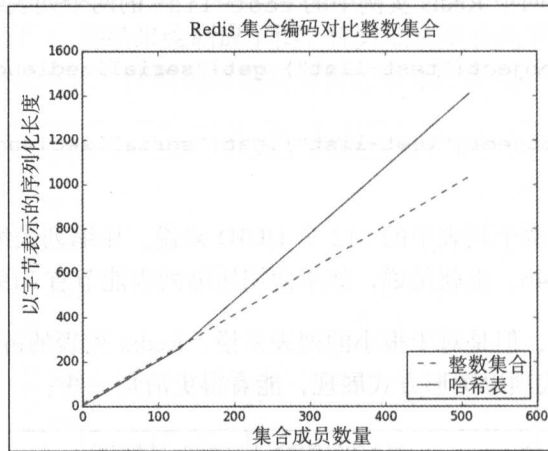
虽然没有那么明显，但是对于很小的列表来说，Redis 列表的链表编码实际上更高效。如果我们将前 50 个列表项以图形方式展现，能看得更清楚一些：



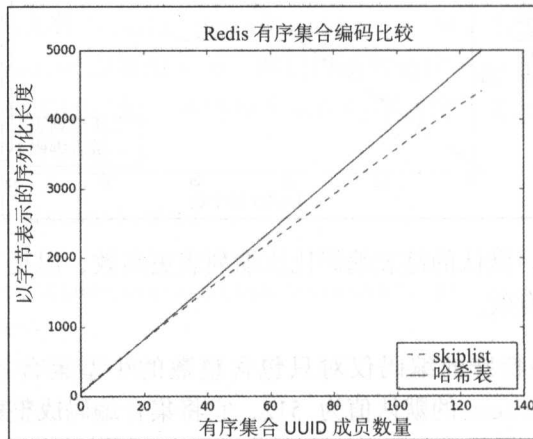
对于小型列表来说，默认的链表编码比压缩列表更高效，但是当列表的大小增长时，压缩列表编码将变得更有效。

对集合来说，这些特殊的编码仅对只包含整数的小型集合才有优势。Redis 指令 set-max-intset-entries 的默认值为 512，它将集合编码成整数集合数据类型。运行 Python 函数对两个 Redis 实例进行实验，获取结果如下所示：

```
>>>INSTANCE1.debug_object("test-set").get("serializedlength")
1034
>>>INSTANCE2.debug_object("test-set").get("serializedlength")
2874
```



最后，在对有序集合特殊编码的研究和实验中，我们将使用两个完全一样的 Redis 实例，并将其中一个实例的 `zset-max-ziplist-entries` 设置为 0，以强制 Redis 使用哈希表作为数据编码格式。对于每一个 `ZADD` 命令来说，`plot_sortedset` 函数将向每个实例中添加 UUID，同时将分值设置为 0，以便以 UUID 的字母顺序进行排序。



从 Python shell 上比较两个实例中 `test-sorted-set` 的结果，如下所示：

```
>>>INSTANCE1.debug_object("test-sorted-set").get("serializedlength")
4421
>>>INSTANCE2.debug_object("test-sorted-set").get("serializedlength")
4994
```

对于压缩列表实现, `test-sorted-set` 的序列化长度为 4,421 字节, 而 `skiplist` 实现的序列化长度为 4,994, 要优于压缩列表 573 个字节。

对所有的压缩列表实现来说, 随着数据结构的大小逐渐增加, 其计算花费的时间也将成倍增加。对基于 Redis 的应用程序来说, 这些阈值调整关乎内存大小与性能之间的平衡。需要认识到的是, 大型的 `ziplist` 数据结构在同等价 Redis 数据类型的默认数据结构相比时会变得缓慢。

把位、字节和 Redis 字符串用作随机访问数组

在第 1 章和第 2 章中, 我们讨论了 Redis 中位图的使用, 包括各种各样的命令, 例如 `SETBIT`、`GETBIT`、`BITCOUNT`、`BITPOS` 和 `BITOP`。为了展现使用位图对比集合所节省的内存, 让我们回顾之前的茶示例, 来看看我们如何表示茶里是否含有咖啡因。这可是关乎部分读者尝试醒来还是入眠的重要决定! 使用一个名为 `tea:caffeinated` 的 Redis 集合就能轻松解决该问题。我们可以通过使用 `SADD` 命令将所有包含咖啡因的茶的键添加到该集合中。为了更好地演示, 我们假设有超过 10,000 种茶, 其中超过 60% 的茶含有某种可追溯级别的咖啡因。我们将沿用最初的 Redis 键模式。同时, 由于我们为每种茶使用了唯一递增的计数器, 因此可以在 `redis-cli` 上对 `tea/caffeinated` 集合进行如下方式的填充(假设 `tea:4` 是类似绿茶的茶):

```
127.0.0.1:6379> SADD teas/caffeinated tea:1 tea:4
(integer) 2
```

为了模拟完整的 10,000 种茶, 我们将使用 Python 模块 `small_types_tests.py` 中的函数 `populate_tea`:

```
def populate_tea(full=True):
    for i in range(10000):
        if random.random() <= .6:
            member = i
            if full:
                member = "tea/{}".format(i)
            INSTANCE1.sadd("teas/caffeinated", member)
            INSTANCE1.setbit("teas/caffeine", i, 1)
```

在 `populate_tea` 函数中, 我们调用了 `random.random()` 方法来生成 0 到 1 中的随机值。为了能满足 60% 的茶含有咖啡因的假设, 我们会检测该随机数是否小于 0.6, 并将该

含有咖啡因的茶添加到 `teas:caffeinated` 集合和 `teas:caffeine` 位图中。运行此函数后，我们会有超过 10,000 种茶：

```
>>>INSTANCE1.debug_object("teas/caffeinated").get("serializedlength")
53879
>>>INSTANCE1.debug_object("teas/caffeine").get("serializedlength")
1252
```

为了查看仓库中有多少茶含有咖啡因，我们可以在集合 `teas/caffeinated` 上运行 `SCARD` 命令，或者在位图 `teas/caffeine` 上运行 `BITCOUNT` 命令：

```
127.0.0.1:6379> SCARD teas/caffeinated
(integer) 6063
127.0.0.1:6379> BITCOUNT teas/caffeine
(integer) 6063
```

对于我们的集合 `teas/caffeinated` 来说，其序列化长度为 53,879 字节，而存储着原始字符串的位图的序列化长度为 1,252 字节，显著节省了 52,627 字节的内存！现在，你可能会疑惑为什么我们的 `populate_tea` 函数有 `full` 这个参数呢？在最初实现之中，我们为每种茶存储了字符串键。为了减少 `teas:caffeinated` 集合的大小，我们将函数参数 `full` 设置为 `False`，这样我们就只存储茶计数器的整数值。因此，通过设置 `full=False` 运行 `populate_tea` 函数，此时从 `redis-cli` 中获得 `teas/caffeinated` 和 `teas/caffeine` 的长度分别为：

```
>>>INSTANCE1.debug_object("teas/caffeinated").get("serializedlength")
17800
>>>INSTANCE1.debug_object("teas/caffeine").get("serializedlength")
1252
```

在集合 `teas/caffeinated` 中通过使用整数代替字符串，我们能将集合的大小显著地从 52,879 字节降至 17,800 字节，但是位图 `teas/caffeine` 仍然要比集合小一个数量级。然而，位图不是万能的。稀疏位图中数以百万计的空间浪费了，每个偏移量中的首个位被设置，而其余的位都未被使用。同时，Redis 整数集合和哈希提供了额外的功能，这些功能如果使用位图来实现非常困难或者需要大量的客户端代码才行。

优化哈希，高效存储

在有关 Redis 的内存优化方面，Salvatore Sanfilippo 使用 Redis 哈希实现了内存非常高

效的高级键值数据存储。为了展示如何使用这项技术，我们将回到第 1 章中的 MARC 记录的传统表达，并比较两种存储 MARC JSON 序列化到 Redis 的方式。在第一个实验中，我们将简单地使用一对一的方式，即 MARC redis 键对应字符串方式存储的 JSON 序列化的 MARC 记录。对于第二个实验来说，我们将使用哈希存储相同的 JSON 序列化记录。

对于这两个实验来说，我们的 Redis 模式将使用 marc 作为前缀，使用分号作为分隔符，之后是唯一递增的计数器，总的来说是一个简单而且常见的 Redis 模式：

```
marc:25
marc:334
marc:8990
marc:122345
```

我们的数据集是刚好超过 17,000 的 MARC21 记录。这些记录是由私立文科学院小型高校图书馆中的读者借阅的最为流行的资料。在第一个实验中，用于 MARC 记录的摄取算法的实现是下列 basic_ingestion 函数中的三行代码：

```
def basic_ingestion(record):
    """Function takes a MARC record, converts it into JSON, and
    then saves the result as string in Redis.
    Args:
        record -- MARC21 record
    """
    marc_json = record.as_json()
    redis_key = "marc:{}".format(INSTANCE1.incr("global:marc"))
    INSTANCE1.set(redis_key, marc_json)
```

要在 Python 命令行上运行该函数，还需要从 pymarc.MARCReader 生成器读取 MARC 记录的 popular_records Python 列表开始：

```
>>>marc_reader = pymarc.MARCReader(
    open('tutt-library-popular.mrc', 'rb'), to_unicode=True)
>>> for record in marc_reader:
    base_ingestion(record)
```

在提取这些 MARC 记录之后，我们的 Redis 数据库的基本信息如下所示：

```
127.0.0.1:6379> DBSIZE
(integer) 17145
```

```
127.0.0.1:6379> INFO memory
# Memory
used_memory:58283440
used_memory_human:55.58M
used_memory_rss:58118144
used_memory_peak:58283440
used_memory_peak_human:55.58M
used_memory_lua:35840
mem_fragmentation_ratio:1.00
mem_allocator:libc
```

我们的 Redis 数据库包含了 17,145 个键,每个键存储了 MARC 记录的 JSON 表达。Redis 数据库大小为 55.58MB。现在,我们使用基于 Redis 哈希的方法存储相同的序列化 JSON 的 MARC 记录。

首先,我们使用简单的算法将键分成两部分:第一部分用作键,同时最后的两个字符作为哈希中的字段名。进一步讲,我们约定所有的 `marc:{record-number}` 键以数字结尾。基于 Redis 模式的实现方式 (Redis 对象映射器,客户端验证代码,或是外部模式验证器),我们可以从 `marc_hash.py` 模块中的代码片段看到验证被强制执行。`marc_hash.py` 模块可以从本书的网站或者对应的 Github 仓库 <https://github.com/jermnelson/marc-redis> 下载。

```
def split_key(redis_key):
    """
    new_key, field = redis_key[:-2], redis_key[-2:]
    if not new_key.startswith('marc'):
        raise InvalidKeyError(redis_key, "Must start with marc")
    try:
        int(field)
    except ValueError:
        raise InvalidKeyError(redis_key, "Last two characters must
        be integers")
    return new_key, field
```

在定义了该函数之后,我们将定义第二个函数,它接收 MARC21 记录作为参数,将其序列化为 JSON,然后使用 Redis 实例递增全局 MARC 计数器,并进行 Python 字符串格式化。

```
def hash_ingestion(record):
    marc_json = record.as_json()
    redis_key = "marc:{}".format(REDIS.incr("global:marc"))
```

`split_key` 函数接收现存的 Redis 键并返回由 `global:marc` 计数器最后两位组成的新的字段名，同时字段对应的值为序列化 json 格式。

```
key, field = split_key(redis_key)
REDIS.hset(key, field, marc_json)
```

在定义了这些函数之后，我们将使用 MARC 记录集合运行测试实验，结果如下所示：

```
127.0.0.1:6380> DBSIZE
(integer) 174
127.0.0.1:6380> INFO memory
# Memory
used_memory:57823456
used_memory_human:55.14M
used_memory_rss:57851904
used_memory_peak:57836688
used_memory_peak_human:55.16M
used_memory_lua:35840
mem_fragmentation_ratio:1.00
mem_allocator:libc
```

使用该替代方法，存储 17K+ 的 MARC 记录只用了 174 个键，内存占用 55.14MB，比使用基本字符串实现节约了 431,536KB³。

硬件和网络延迟

在应用程序中，性能问题很容易被误认为是 Redis 数据库内存不足造成的。其实该问题很有可能是有关客户端应用程序和后端服务器之间硬件或者网络延迟的问题。以 Redis 社区对延迟的理解，可以分为以下三种。

- **命令延迟**：指的是执行一条命令所花费的时间。一些命令执行起来非常快速，时间复杂度为 $O(1)$ ，而另一些命令的时间复杂度为 $O(n)$ ，有可能是导致这类延迟的源头。

3 译者注：根据两个例子中的数据比较得出，实际节约的内存应为 459984。

- **往返 (round) 延迟**: 当客户端发送命令并在之后接收来自 Redis 服务器的响应之间所花费的时间可能是由于网络拥塞造成的。
- **客户端延迟**: 假如多个客户端同时尝试连接到 Redis, 可能会导致并发延迟, 这时较晚到达的客户端请求会在队列中排队以等待较早到达的客户端请求处理完成。

为了协助调试问题, Redis 有一个特殊的模式用来监控命令延迟, 可以通过设置 `redis.conf` 或者发送 `CONFIG SET` 设置 `latency-monitor-threshold` 指令。该指令设置了以毫秒为单位的限制, 超过该限制的所有或部分命令及 Redis 实例的活动 (称作时间) 都会被记录下来。该指令默认值为 0, 意味着 Redis 不会自动运行延迟监控, 因此我们必须主动设置。借鉴 Redis 官方文档中对延迟监控的示例, 我们首先将 `latency-monitor-threshold` 指令设置为 100 毫秒:

```
127.0.0.1:6379> CONFIG SET latency-monitor-threshold 100
```

现在, 我们运行一些 `DEBUG SLEEP` 命令演示 Redis 延迟监控的众多子命令与功能。

```
127.0.0.1:6379> DEBUG SLEEP 1
127.0.0.1:6379> DEBUG SLEEP .25
127.0.0.1:6379> LATENCY LATEST
1) 1) "command"
   2) (integer) 1433877394
   3) (integer) 250
   4) (integer) 1000
```

命令与子命令返回超过延迟阈值的最新的 Redis 命令, 包括事件名称、延迟事件发生时的 UNIX 时间戳, 以毫秒为单位的最新事件延迟, 以及历来该事件的最大延迟时间。在我们的示例中, 行 2) 1433877394 指的是最新 `DEBUG SLEEP` 命令的时间戳, 行 3) 250 指的是 `DEBUG SLEEP .25` 命令的结果, 最后, 行 4) 1000 记录了我们第一次调用的 `DEBUG SLEEP 1` 命令。

`LATENCY HISTORY` 命令与子命令返回追踪的最新 160 个延迟事件。从 `redis-cli` 上运行该命令得到以下结果:

```
127.0.0.1:6379> LATENCY HISTORY command
1) 1) (integer) 1433877379
   2) (integer) 1000
2) 1) (integer) 1433877394
   2) (integer) 250
```

结果是由 UNIX 时间戳和每个事件花费的以毫秒为单位的时间所组成的元组。

LATENCY 命令与 RESET 子命令可以用来清除所有的延迟事件, 或者通过指定一到多个事件名称清除选定的事件。LATENCY GRAPH 命令将生成一个 ASCII 风格的图, 用来表示自从最近一次 LATENCY RESET 命令以来所有记录在案的延迟事件。redis-cli 上执行的结果如下所示 (为了简明起见删除了部分返回值):

```
127.0.0.1:6379> DEBUG SLEEP .5
OK
(0.50s)
127.0.0.1:6379> DEBUG SLEEP .3
127.0.0.1:6379> DEBUG SLEEP .8
127.0.0.1:6379> DEBUG SLEEP .2
127.0.0.1:6379> DEBUG SLEEP .6
127.0.0.1:6379> LATENCY GRAPH command
command - high 800 ms, low 201 ms (all time high 800 ms)
-----
-----
#
_ | o
| | |
|o|_|

55544
95170
sssss
```

最后, LATENCY DOCKER 模式提供了丰富的、人类可读的 (闪烁着由斯坦利·库布里克指导的《2001 太空漫游》中 HAL 9000 的光芒!) 统计数据, 例如延迟峰值之间的平均时间, 峰值的中位数偏差, 以及人类可以理解的延迟分析与减少延迟的建议。

操作系统建议

Redis 在支持 POSIX 的操作系统上开发并运行大部分应用程序。这些操作系统包括 Linux 及其发行版、Macintosh OS 和其他 BSD 派生的操作系统, 以及其他商业 UNIX 操作系统。Redis 项目并不官方支持 Microsoft Windows, 不过 Microsoft 的 Open Tech 团队开发

并维护了一个基于 64 位 Windows 的版本。还有不少针对 Raspberry Pi4 和 Android5 平台上运行 Redis 的实验。

在 Linux 上运行 Redis 的建议



你应当通过运行下列脚本禁用内核中的透明大页（transparent huge pages）：

```
echo never > /sys/kernel/mm/transparent_hugepage/enabled
```

将 `vm.overcommit_memory` 设置成 1，以避免 Linux 虚拟机通过交换的方式进行后台保存时出现问题。

总结

本章从可以调整的内存相关指令开始，然后查看了多种键驱逐策略来应对 Redis 的可用内存限制。接下来研究的是针对小型哈希、列表、有序集合及特定情况下（集合下的不同的内存高效编码。之后，我们看到了将位图字符串用作随机存取数组，以及更高效地将哈希用作内存高效的高级键值存储。最后，我们查看使用 Redis 延迟监控模式来追踪有问题的、运行时间较长的 Redis 命令，并给出在 Linux 上运行 Redis 的几点建议。第 4 章我们将注意力集中在软件开发上，同时开启了 Redis 的 C 源码之旅，然后再使用三种不同的编程语言使用 Redis 客户端。

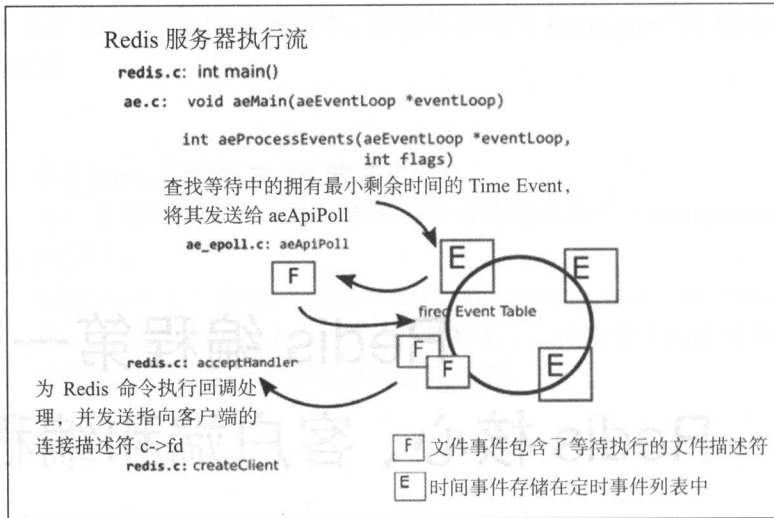
4

Redis 编程第一部分： Redis 核心、客户端和编程语言

本章将从 Redis C 源码开始讲起，研究主要的 Redis C 头文件和代码文件是如何相互关联并协同工作的。之后将在较高的层面上介绍代码执行流程，从而使我们能实现自定义 Redis 命令。然后，为了准备好在第 5 章使用两种不同编程语言的 Redis 客户端，我们将对 **Redis 转储二进制（RDB）** 格式进行详细地研究。Redis 用此二进制格式来持久化数据库快照到磁盘上。同时，我们也将介绍 Redis 协议规范，它是一种被客户端用来和 Redis 服务器通信的底层通信格式。最后，我们将在不同用例场景中使用这些 Redis 客户端和编程方法。

Redis 的内部结构

作为网络服务器，Redis 内部操作遵循基本的执行流程，服务器端等待并在特定端口上监听请求连接。如果到达的客户端使用正确的 **Redis 序列化协议（RESP）** 语法和格式，服务器就会接受此连接。在接受套接字连接后，Redis 会为非阻塞读和写操作在数据库的内存状态上产生一个描述符。



对于 Redis 服务器来说, main 函数通过调用 aeMain 函数创建事件循环, 而该函数会创建一个无限 while 循环。该循环测试事件循环的 stop 属性, 并在测试失败时退出循环。aeMain 中 while 循环的每次迭代都会调用 aeProcessEvents 函数, 并传入事件循环的指针及标志位。aeProcessEvents 函数在处理所有文件事件之前会处理所有基于时间的事件。别忘了, POSIX 系统将运行中的进程视为文件描述符, 因此即便是从内存中读取值, Redis 也会将这些读取操作视为事件管理代码中的文件描述符。在 Redis 中, 基于时间的事件会根据传递给 aeProcessEvents 函数的标志位控制事件循环对事件的处理。事件范围从立即, 到尽可能最短时间, 再到阻塞, 以及永远等待, 所有这些可以使用时间值结构进行设置。

在运行中的 Redis 实例上使用 LLDB 调试器, 我们可以通过查看 Redis 会话中的每一帧来追踪到目前为止的执行流。在以下回溯中, 我们从最近的帧开始:

```
(lldb) thread backtrace
```

```
* thread #1: tid = 0xf41e, 0x000000001000047ef redis-
server`aeProcessEvents [inlined] aeGetTime(milliseconds=<unavailable>) +
8 at ae.c:186, queue = 'com.apple.main-thread', stop reason = step in
```

由于 Redis 是单线程服务器应用程序, 我们从 thread #1 开始, 并将 Redis 服务器停止在预先定义的断点处。沿着最近的帧向后调试, 我们将追踪执行路径的历史, 回到最初的 frame #5。

```
* frame #0: 0x00000001000047ef redis-server`aeProcessEvents [inlined]
aeGetTime(milliseconds=<unavailable>) + 8 at ae.c:186
```

如之前的代码所示,从最新的 frame #0 开始,aeGetTime 函数是被 frame #1 中的当前 aeEventLoop 结构调用的。

```
frame #1: 0x00000001000047e7 redis-server`aeProcessEvents + 108 at
ae.c:304
```

frame #1 处于 processTimeEvents 函数的 while 循环中,它会从 frame #0 调用 aeGetTime 函数,并传入两个时间引用参数。processTimeEvents 被 frame #2 中的 aeMain 函数调用。

```
frame #2: 0x000000010000477b redis-server`aeProcessEvents(eventLoop=
0x0000000100323150, flags=<unavailable>) + 651 at ae.c:423
```

在 frame #2 中,aeProcessEvents 函数和当前指向 eventLoop 的参数,在 aeMain 中被调用。aeMain 的返回值最终将会递增 aeProcessEvents 函数中的 processed 变量。

```
frame #3: 0x0000000100004a1b redis-server`aeMain(eventLoop=
0x0000000100323150) + 43 at ae.c:455
```

在 frame #3 中,我们处于 aeMain 函数中的 while 循环。该函数在调用时被传入指向在 frame #4 中创建的当前 eventLoop 结构的指针。

```
frame #4: 0x000000010000f1a8 redis-server`main(argc=<unavailable>,

```

在 frame #4 中,main 函数起始于 redis.c 的第 3892 行,之后调用了 ae.c 中的 aeMain 函数:

```
argv=0x00007fff5fbffb20) + 1256 at redis.c:3892
```

```
frame #5: 0x00007fff900185c9 libdyld.dylib`start + 1
```

在 frame #5 中,我们看到 Redis 服务器是由操作系统启动的。

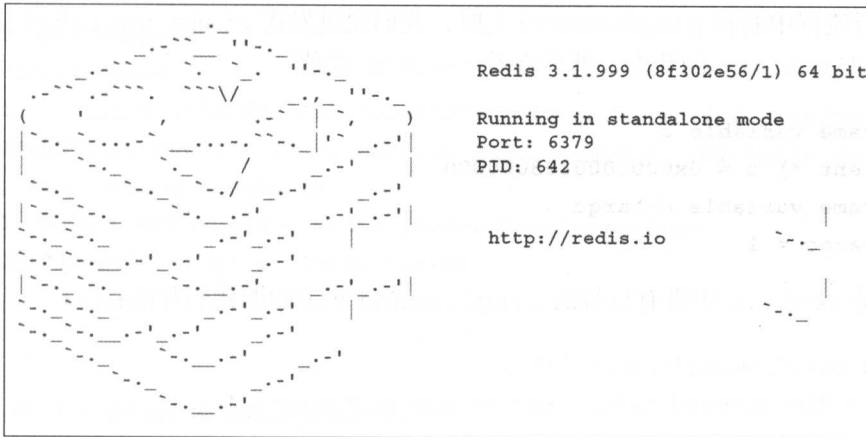
aeEventLoop 结构包含两个重要的结构:包含事件循环注册事件的 aeFileEvent,以及指向 aeFiredEvent 结构的指针。aeFiredEvent 结构包含两个变量:其中一个是文件描述符,另一个位掩码用于描述该事件。在 aeProcessEvents 函数中,每个 FileEvent 通过 aeApiPoll 函数调用 Linux 的 epoll_wait 系统函数,该函数会在文件

描述符上等待 I/O 活动。这是一个阻塞式调用。aeApiPoll 的实现代码在 ae_poll.c 中。当在 aeFileEvent 上发生 I/O 读写操作时，aeProcessEvents 最终向 aeFiredEvent 中添加一条额外的条目。

call 函数的第一个参数是指向 redisClient 的指针和 flags 整数。在 call 函数中，命令在被执行前会发往任何处于监控模式下的客户端。之后，该命令会通过调用 redisClient 结构的 proc 函数执行。现在该执行流依赖于接收到的 Redis 命令。最简单的 Redis 命令之一是 PING 命令。该命令将导致服务器重放 PONG 响应。PING 是以 redis.c 中的 pingCommand 函数实现的。

从 Redis 的根目录运行 LLDB 调试器，我们将启动 Redis 的运行实例，并在 pingCommand 上设置断点：

```
$ lldb
(lldb) file src/redis-server
Current executable set to 'src/redis-server' (x86_64).
(lldb) breakpoint set -f redis.c -l 2482
Breakpoint 1: where = redis-server`pingCommand + 9 at redis.c:2484,
address = 0x0000000100007639
Issuing the run command in our debugging session will launch Redis
server, getting the ascii screen of Redis.
(lldb) run
Process 642 launched: '/Users/jeremynelson/redis-dev/src/redis-server'
(x86_64)
642:C 08 Jul 06:54:17.092 # Warning: no config file specified, using the
default config. In order to specify a config file use
/Users/jeremynelson/redis-dev/src/redis-server /path/to/redis.conf
642:M 08 Jul 06:54:17.093 * Increased maximum number of open files to
10032 (it was originally set to 2560).
```



```
642:M 08 Jul 06:54:17.097 # Server started, Redis version 3.1.999
642:M 08 Jul 06:54:17.103 * DB loaded from disk: 0.006 seconds
642:M 08 Jul 06:54:17.103 * The server is now ready to accept connections
on port 6379
```

下一步，我们将打开另一个终端窗口，启动 `redis-cli`，发送 PING 命令：

```
$ src/redis-cli
127.0.0.1:6379> PING
```

切回到调试器终端窗口，我们会看到以下输出内容：

```
Process 642 stopped
* thread #1: tid = 0x18cff, 0x00000000100007639 redis-server`pingComman
d(c=0x00000000103000000) + 9 at redis.c:2484, queue = 'com.apple.mainthread',
stop reason = breakpoint 1.1
frame #0: 0x00000000100007639 redis-server`pingCommand(c=
0x00000000103000000) + 9 at redis.c:2484
    2481 * in Pub/Sub mode. */
    2482 void pingCommand(redisClient *c) {
    2483 /* The command takes zero or one arguments. */
-> 2484 if (c->argc > 2) {
    2484         addReplyErrorFormat(c, "wrong number of arguments for
's' command",
    2486             c->cmd->name);
    2487         return;
```

当执行流程中断在 pingCommand 上时, 我们通过发送 frame variable LLDB 命令检测 redisClient 的状态, 然后查看 c->argc 的值:

```
(lldb) frame variable c
(redisClient *) c = 0x00000000103000000
(lldb) frame variable c->argc
(int) c->argc = 1
```

以下是 redis.c 中带有行号的 pingCommand 相关的源代码片段:

```
2481 void pingCommand(client *c) {
2482     /* The command takes zero or one arguments. */
2483     if (c->argc > 2) {
2484         addReplyErrorFormat(c, "wrong number of arguments for '%s' command",
2485             c->cmd->name);
2486         return;
2487     }
2488
2489     if (c->flags & CLIENT_PUBSUB) {
2490         addReply(c, shared.mbulkhdr[2]);
2491         addReplyBulkCBuffer(c, "pong", 4);
2492         if (c->argc == 1)
2493             addReplyBulkCBuffer(c, "", 0);
2494         else
2495             addReplyBulk(c, c->argv[1]);
2496     } else {
2497         if (c->argc == 1)
2498             addReply(c, shared.pong);
2499         else
2500             addReplyBulk(c, c->argv[1]);
2501     }
2502 }
```

从代码的 2483 行开始, 我们可以看到 redisClient 判断 if (c->argc > 2) 条件时失败了。

我们同样可以通过发送 thread 回溯命令检查执行记录。我们将忽略已经检查过的帧:

```
(lldb) thread step-in
Process 642 stopped
* thread #1: tid = 0x18cff, 0x000000010000765f redis-server`pingCommand(c=0x0000000103000000) + 47 at redis.c:2490, queue = 'com.apple.main-thread', stop reason = step in
frame #0: 0x000000010000765f redis-server`pingCommand(c=0x0000000103000000) + 47 at redis.c:2490
2486         return;
2487     }
2488
-> 2489 if (c->flags & REDIS_PUBSUB) {
2490     addReply(c,shared.mbulkhdr[2]);
2491     addReplyBulkCBuffer(c,"pong",4);
2492     if (c->argc == 1)
```

跳过行 2489-2496, 我们可以看到如下输出 (为清晰起见, 省略了部分输出):

```
(lldb) thread step-over
.
.
.
2497     } else {
-> 2498 if (c->argc == 1)
2499     addReply(c,shared.pong);
2500     else
2501     addReplyBulk(c,c->argv[1]);
(lldb) frame variable c->argc
(int) c->argc = 1
```

为了证实我们的想法, 我们展示了变量 `c->argc` 的值为 1, 因此该条件判断之后会执行 `addReply` 函数。然后, `addReply` 函数会运行 `prepareClientToWrite` 函数。`prepareClientToWrite` 函数会在我们期望向客户端发送新数据时被调用, 并伴随着 `REDIS_OK` 返回值及向 `eventLoop` 写入数据的写处理器套接字。执行流会在 `prepareClientToWrite` 函数结束时返回 `call` 函数。在该命令完成处理后, `call` 函数将返回到 `processCommand` 函数。之后 `processCommand` 函数会将 `REDIS_OK` 或 `REDIS_ERR` 状态返回给 `processInputBuffer` 函数, 从而调用 `resetClient` 命令。在客户端被重置后, `processInputBuffer` 函数将控制权交还给 `aeProcessEvents` 函数。

最后，控制权被交还给运行中的 Redis 服务器实例的 aeMain 事件循环。

理解 redis.h 和 redis.c

Redis 作为开源项目的一个非常显著的优势是你可以下载、研究并实验它的 C 源代码。我们对 Redis 源代码的研究始于 redis.h 和 redis.c 这两份代码文件。它们包含了运行服务器的主要源代码。

在接下来的几个段落里，我们将首先研究定义在 redis.h 里的常量、结构和宏。

在 redis.h 文件的开始部分，includes 指令从 C 标准库中导入了诸如 stdio.h、stdlib.h 和 time.h 的头文件，同时也从依赖库中导入了其他头文件，例如用于 Linux 线程的 pthread.h、syslog.h 和 lua.h。redis.h 文件中的下一部分定义了 mstime_t 毫秒时间类型为 64 位有符号长整型，范围从 -9,223,372,036,854,775,807 到 +9,223,372,036,854,775,807。但是，由于我们存储的是毫秒，负的时间值是无效的，因此没有在 Redis 服务器或者客户端代码中用到。在定义了 mstime_t 之后，本地头文件定义了用于运行和管理 Redis 实例的各种辅助宏、函数和 API 函数接口。这些头文件包括用于 Redis 的事件库 (ae.h)、动态安全字符串 (sds.h)、哈希表 (dict.h)、链表 (adlist.h)，以及 malloc 的一个感知总共可用内存的版本 (zmalloc.h)。这里定义了第 3 章介绍的用于网络通信 (anet.h)、压缩列表 (ziplist.h) 和整数集合结构 (intset.h) 的头文件，以及 version.h、util.h、latency.h 和 sparkline.h 头文件。

在这些 include 语句之后，redis.h 将常量 REDIS_OK 定义为 0，同时将常量 REDIS_ERR 定义为 -1。这两个常量被广泛用于 Redis 项目中 Redis 命令的实现代码之中。redis.h 头文件的下一部分定义了服务器默认配置，在缺少 Redis 配置文件时使用。举例来说，REDIS_SERVERPORT 定义为 6379，连接到 Redis 的客户端最大数量 REDIS_MAX_CLIENTS 定义为 10,000，以及 REDIS_DEFAULT_RDB_FILENAME 定义为 dump.rdb。其他 look-up、metrics 及 I/O 相关的定义遵循 Redis 默认配置的值，并在 redis.h 中以 Redis 命令标志的方式罗列。

这些在 redis.c 代码文件中的命令标志是 redisCommand 结构的成员。该结构被传递给 processCommand 函数，我们会在稍后章节中讨论。所有的 Redis 命令 (GET、SET、HSET 等) 都存储在 redisCommandTable 中。redisCommandTable 是 C 语言的结构，定义在 redis.c 中。在 redisCommandTable 中，每个命令处在单独的行上，并包含特定于该 Redis 命令的一些设置。

每一行的命令拥有下列字段(以从左到右的顺序):

- 命令名称: 命令的小写字符串名称。
- 函数指针: 指向该命令实现的函数指针。
- 函数参数数量: 命令函数期望的参数总数。
- 函数 sflags: sflags 字段包含了所有不同 Redis 命令字段的字符串。
- 前一个字段(sflags)的字符串值, 是由 redis.h 中定义的常量计算得来的位掩码。
- 可选的函数, 用于提取并且/或者计算命令的键参数。
- 传递给 Redis 命令的参数列表的下标, 即该数据结构值的 Redis 键。
- 参数中最后一个键的下标, 这样就可以允许 Redis 命令每次操作多个键。
- 以微秒为单位的命令总共执行的时间。该值由 Redis 计算得出, 应该始终设置为 0。
- 该命令总共被调用多少次。该值是可变的, 在运行时得出, 应始终设置为 0。

下面是 GET 和 SET 两个命令的例子:

```
{"get",getCommand,2,"rF",0,NULL,1,1,1,0,0},
{"set",setCommand,-3,"wm",0,NULL,1,1,1,0,0},
```

示例中第一个字段是小写的字符 get 和 set。第二个字段分别是指向 getCommand 和 setCommand 函数的指针。第三个字段是参数个数, 对 getCommand 来说为 2, 对 setCommand 来说是-3, 意味着参数的总数大于或者等于 3。

对于第四个字段, 在 GET 命令中为 rF 标志, 其中 r 代表该命令为读命令, F 代表该命令是快速命令。这意味着其时间复杂度为 $O(1)$ 或 $O(\log(N))$ 。对于快速命令来说, 如果内核调度器继续为运行 Redis 服务器提供运行时间, 那么 Redis 服务器事件循环永远不应该延迟该命令的执行(对我们的示例来说, 就是 GET 命令)。对 SET 命令的标志来说, w 意味着该命令是一个写命令, m 意味着该命令会增加内存使用并在 Redis 耗尽内存时禁止使用。此外, 还有 12 个其他命令标志, 例如代表随机的 R、代表排序命令的 S、用于禁止命令在脚本中调用的小写 s。

第五个字段从前一个字段(sflags)获取字符串值并为 rF 和 wm 计算位掩码。第六个字段在这两个命令中均设置成了 NULL。这意味着 GET 和 SET 均不需要可选函数来获取键参数。第七个字段为参数的下标。这就是键, 而且对于 GET 和 SET 来说值均为 1。第七个字段和第八个字段均为 1, 因此 getCommand 和 setCommand 都只接受 1 个键, 即参数列表中的第一个参数。第九个字段和第十个字段值均设置为 0, 这是因为这些字段是动态的, 由运行中的 Redis 通过计算得出。

我们之前就遇到的函数 `redis.c` 头文件中的 `processCommand` 函数就首先用到了这些常量。`processCommand` 函数接收指向 `redisClient` 结构的指针。该结构是 Redis 的基石，它描述了服务器上内部 Redis 进程与外界通信的状态。`networking.c` 中的 `createClient` 函数接收单个参数 `int fd`、分配内存并将指针存储到结构中。之后，`createClient` 继续初始化所有重要的变量，例如指向当前活跃的 Redis 数据库(`redisDb`)的指针。我们也会从 `redis.h` 中的 `redisClient` 挑选出这几行重要的属性讲解：

```
529 typedef struct redisClient {
530     uint64_t id;           /* Client incremental unique ID. */
531     int fd;
532     redisDb *db;
```

行 530 定义了唯一 64 位整数 ID `t_id`，行 531 是文件描述符整数 ID，行 532 是指向 Redis 数据库的指针：

```
537     int argc;
538     robj **argv;
```

行 537 中的 `argc` 指的是命令的参数总数，行 538 中的 `argv` 引用的是命令的返回值：

```
539     struct redisCommand *cmd, *lastcmd;
```

行 539 定义了指向当前的 `redisCommand` 结构的指针 `*cmd`，以及指向之前的 `redisCommand` 结构的指针 `*lastcmd`。

```
550     int flags;           /* REDIS_SLAVE | REDIS_MONITOR | REDIS_
MULTI ... */
```

行 550 是 `*flags` 位掩码，包含了 Redis 服务器的操作模式：

```
554     int repldbfd;        /* replication DB file descriptor */
555     off_t repldboff;     /* replication DB file offset */
556     off_t repldbsize;    /* replication DB file size */
```

行 554、555 和 556 是主从数据库文件变量，用于从主节点复制到对应的所有从节点。

```
564     char replrunid[REDIS_RUN_ID_SIZE+1]; /* master run id if this
is a master */
565     int slave_listening_port; /* As configured with: SLAVECONF
listening-port */
```

如果运行中的 Redis 实例是主节点, 那么行 564 即为主节点的运行 ID。行 565 是 Redis 从实例监听的端口号。

```
571     list *watched_keys;      /* Keys WATCHED for MULTI/EXEC CAS */
572     dict *pubsub_channels; /* channels a client is interested in
(SUBSCRIBE) */
573     list *pubsub_patterns; /* patterns a client is interested in
(SUBSCRIBE) */
```

如果 Redis 实例处于 multi/exec 模式下, 那么行 571 代表指向 watched_keys 链表的指针。行 572 和 573 均为指向链表的指针, 在 Redis 处于发布/订阅模式时, 分别存储了订阅的信道和模式。

```
576     /* Response buffer */
577     int bufpos;
578     char buf[REDIS_REPLY_CHUNK_BYTES];
```

行 577 和 578 存储了服务器执行命令的 RESP。

processCommand 函数首先检测 quit 命令是否被发往服务器, 如果是的话就返回 REDIS_ERR (对 quit 命令的处理是在另一个函数中进行的)。在 processCommand 中进行的额外错误检测是用来检查命令是否存在或该命令的参数个数是否有误。该函数并不会为这些错误返回 REDIS_ERR, 而是使用指向命令的指针及错误消息调用 addReplyErrorFormat, 然后返回 REDIS_OK 状态。这是因为 Redis 服务器对于请求的客户端来说仍然是正常运作的。如果命令要求认证而客户端没有获取服务端认证, 就会使用未认证消息调用 addReply 函数并返回 REDIS_OK。遵照认证要求, processCommand 函数将命令重定向到集群分区, 或是添加一条错误消息并返回给请求的客户端。

在 processCommand 函数中, 在集群错误检测之后, 函数会在 Redis 服务器内存溢出时检测并应对各种情况。首先, 在尝试通过移除键的方式补充内存之后, 如果没有可用内存, 那么函数就会返回, 同时设置 REDIS_CMD_DENYOOM 常量。该函数会继续进行三个不同的检查来确认是否还可以接收写命令。这取决于以下三种情况, 第一种情况 Redis 是从实例 (slave) 并且设置了 min-slaves-to-write, 第二种情况 Redis 服务器持久化到磁盘功能上发生了故障, 第三种情况 Redis 是从实例并且是只读的。

processCommand 函数的下一部分会处理 Redis 实例的特殊运作模式。特别是处于发布/订阅模式下, 这些操作受限的 Redis 中的 SUBSCRIBE/UNSUBSCRIBE 命令。函数还针

对特殊情况做了更多的错误检测。例如，当 Redis 为从实例并且从主实例上断开连接时，会限制只能使用 INFO 和 SLAVEOF 命令。对于特殊情况来讲，当 Lua 客户端脚本运行得太慢时，函数会标记该命令，调用 addReply 函数并传入错误消息，然后返回给调用的客户端。最后在检测了所有错误之后，processCommand 实际上会将该命令以单独的方式执行，或者以 MULTI/EXEC 事务中的一部分的方式执行。

从 1791 行开始，redis.h 定义了五个基本 Redis 对象类型，如下所示：

```
/* Object types */
#define REDIS_STRING 0
#define REDIS_LIST 1
#define REDIS_SET 2
#define REDIS_ZSET 3
#define REDIS_HASH 4
```

我们在 Redis 源代码中的其他地方看到对象类型赋给 Redis 对象，现在就知道了这个对象类型是整数范围的 0 到 4。在这些 Redis 对象类型之后是 Redis 对象编码类型，例如 Redis 的哈希表、链表、压缩列表和整数集合，分别用 0 到 8 表示。这意味着我们可以组合对象类型和编码类型来反映 Redis 键背后存储的真实数据结构。也就是说，小型哈希值为 REDIS_HASH，并且编码类型为 REDIS_ENCODING_ZIPMAP，分别对应到数字 0 和 3。

接下来是 Redis 持久化模式的大小和其他设定。对于快照 RDB 模式来说，Redis 动态分配位的大小，并在 Redis 键较小时使用键的前两个有效位来收缩大小。根据这两个有效位是否进行了设置，redis.h 定义了四种不同的编码常量。

- **8 位有符号整数：**REDIS_RDB_ENC_INT8 设置为 0
- **16 位有符号整数：**REDIS_RDB_ENC_INT16 设置为 1
- **32 位有符号整数：**REDIS_RDB_ENC_INT32 设置为 2
- **压缩字符串：**REDIS_RDB_ENC_LZF 设置为 3

```
/* When a length of a string object stored on disk has the first two bits
 * set, the remaining two bits specify a special encoding for the object
 * accordingly to the following defines: */
#define REDIS_RDB_ENC_INT8 0 /* 8 bit signed integer */
#define REDIS_RDB_ENC_INT16 1 /* 16 bit signed integer */
#define REDIS_RDB_ENC_INT32 2 /* 32 bit signed integer */
#define REDIS_RDB_ENC_LZF 3 /* string compressed with LZF */
```

对于 AOF 持久化模式来说，定义了三种状态：REDIS_AOF_OFF、REDIS_AOF_ON 及 REDIS_AOF_WAIT_REWRITE。其中 REDIS_AOF_WAIT_REWRITE 代表 Redis 正等待

追加到 AOF 文件中。

针对众多不同类型的 Redis 客户端, `redis.h` 定义了 19 种不同的标志。通过使用位左移的方式, 客户端信息可以用单个字节表示。通过使用位掩码, Redis 命令和其他代码可以快速计算并判定 Redis 是主还是从, 或者客户端是否调用了 `REDIS_MULTI` 而处于 `multi` 执行模式的监控之中, 或者是否使用了 UNIX 域套接字连接上 `REDIS_LUA_CLIENT`。并通过 `REDIS_READONLY` 设置为只读, 或者使用位掩码来计算 `REDIS_PUBSUB` 从而判断客户端是否处于发布/订阅模式。之后, `redis.h` 设置了其他 Redis 客户端常量, 例如客户端阻塞类型、客户端请求类型、客户端类别, 以及以主从双方视角来看待的从实例复制状态。

接下来定义的是四种不同的 Redis 日志级别: `REDIS_DEBUG`、`REDIS_VERBOSE`、`REDIS_NOTICE` 和 `REDIS_WARNING`。通过 `REDIS_DEFAULT_VERBOSITY` 常量设置默认日志级别为 `REDIS_NOTICE`, 还有一个特别的原生日志模式 `REDIS_LOG_RAW`, 它不记录时间戳。

在日志指令这部分中还穿插着 Redis 中使用的不同的数据结构和编码。例如, 有针对 Redis 列表的 `REDIS_HEAD` 和 `REDIS_TAIL`, 分别定义为 0 和 1; 还有针对有序列表的不同的排序选项, 例如 `REDIS_SORT_ASC` 定义为 1, `REDIS_SORT_DESC` 定义为 2, 以及 `REDIS_SORTKEY_MAX` 定义为 1024。在之前的章节中, 我们研究了针对列表、哈希、集合和有序集合的特殊编码下的不同性能权衡, 同时也了解了如何设置像 `hash-max-ziplist-entries` 和 `hash-max-ziplist-value` 这样的指令。这些指令的默认值定义为 `redis.h` 中的 `REDIS_HASH_MAX_ZIPLIST_ENTRIES` 和 `REDIS_HASH_MAX_ZIPLIST_VALUE` 常量。最后, 有关 Redis 集合操作的相关常量有 `REDIS_OP_UNION` 定义为 0, 用于并集操作; `REDIS_OP_DIFF` 定义为 1, 用于差集操作; `REDIS_OP_INTER` 定义为 2, 用于交集操作。

Redis 中不同的缓存策略使用了连续整数进行定义。从 `REDIS_MAXMEMORY_VOLATILE_LRU` 定义为 0 到 `REDIS_MAXMEMORY_NO_EVICTION` 定义为 5。默认的缓存策略 `REDIS_DEFAULT_MAXMEMORY_POLICY` 设置为 `REDIS_MAXMEMORY_NO_EVICTION`。遵循这种定义模式的还有 Lua 脚本超时、单位定义及关闭标识, 请从 `redis.h` 源码文件中查看更多其他定义。

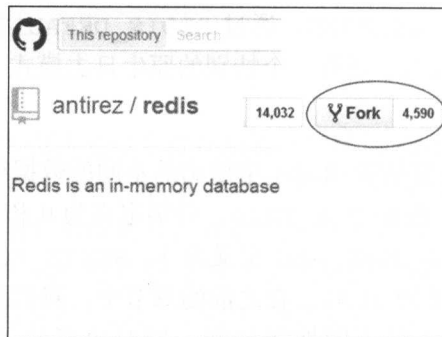
我们跳过了持久化、复制、`multiexec` (事务) 及集群设置, 这是因为我们将在接下来的主题中专门讲解这些不同的 Redis 操作。希望你现在对 Redis 实例的默认值有个大致的了解, 知道可以在 Redis 源码库中进行哪里设置并使用。

准备好！使用 Git 进行 Redis 开发

Salvatore Sanfilippo 将 Redis 发布在三条款 BSD 开源协议下，同时将其托管在 GitHub 上。因此我们能够获取 Redis 的源码备份并创建自己的 Redis 开发版本。首先，我们打开 Linux 系统的命令行，确保我们已经安装了 Git：

```
$ sudo apt-get install git
```

下一步，我们将访问 Redis 在 GitHub 上的仓库 <https://github.com/antirez/redis/>。为了 fork 一份 Redis，我们需要先登录 GitHub（如果没有账户可以创建一个免费账户），然后在页面右上角单击“**Fork**”按钮。如下图所示：



GitHub 上的 Redis Fork 按钮

现在你将被导向到自己的 Redis 复刻版本上。然后你需要按照如下所示将复刻版本克隆到本地（将 jermnelson 替换成自己的 GitHub 用户名）。你应该看到类似如下输出：

```
$ git clone https://github.com/jermnelson/redis.git redis-dev
Cloning into 'redis-dev'...
remote: Counting objects: 20195, done.
remote: Total 20195 (delta 0), reused 0 (delta 0), pack-reused 20195
Receiving objects: 100% (20195/20195), 9.62 MiB | 433.00 KiB/s, done.
Resolving deltas: 100% (13515/13515), done.
Checking connectivity... done.
```

可以通过下列命令将 Redis 复刻版本与主 Redis 仓库保持同步：

```
$ cd redis-dev
$ git remote -v
origin https://github.com/jermnelson/redis.git (fetch)
```

```
origin https://github.com/jermnelson/redis.git (push)
```

下一步,我们将添加上游同步,以便我们将变更从 Redis 主仓库拉到本地:

```
$ git remote add upstream https://github.com/antirez/redis.git
$ git remote -v
origin https://github.com/jermnelson/redis.git (fetch)
origin https://github.com/jermnelson/redis.git (push)
upstream https://github.com/antirez/redis.git (fetch)
upstream https://github.com/antirez/redis.git (push)
```

现在,当想要将变更从上游非稳定分支上拉下来时,会运行以下命令:

```
$ git fetch upstream
```

为了在 Redis 核心仓库上进行开发,我们将创建新的 local-dev 分支:

```
$ git checkout -b local-dev
```

最后,我们将使用下列命令合并所有上游更改:

```
$ git merge upstream/unstable
```

我们的 Redis 复刻版本已经更新了最新的非稳定变更。我们准备好在下一节开始创建新的 Redis C 命令。

练习: 创建自定义 redis 命令

Redis 网站上的教程之一是使用 Redis 的有序集合实现一个自动完成功能。通过 Redis 自动完成方法提升部分单词匹配,我们可以使用英语单词相似音,由大量不同的算法创建发音相似的单词。该方法规范化了用户输入,并将其匹配到索引单词。

我们不会从头开始实现 Lawrence Philip 的双元变音 (double-metaphone) 算法,而是基于之前就已存在的开源项目的 C 源代码创建新的 Redis 命令。该开源项目位于 <https://bitbucket.org/yougov/fuzzy/>。首先,我们将创建两个文件, double_metaphone.h 头文件和 double_metaphone.c C 源文件。当我们想存储一个现存单词的变音时,我们会创建两个新的 Redis 命令: GETDBLMETAPHN 和 SETDBLMETAPHN。SETDBLMETAPHN 命令接受一个 Redis 哈希键和英语单词,将该单词转换为双元变音,并将结果存储为 Redis 哈希中的一个或者两个字段,其值为原来的那个单词。第二个 Redis 命令 GETDBLMETAPHN 接收一对字符串作为参数,将单词转换成变音,并返回最佳匹配的其中一个字段。

SETDBLMEAPHN 的语法如下：

```
"SETDBLMEAPHN <key> <string>"
```

GETDBLMEAPHN 的语法如下：

```
"GETDBLMEAPHN <key>" :
```

为了将这些命令添加到我们的 Redis 中，我们将执行以下步骤：

1. 将 getMetaphone 和 setMetaphone 函数原型添加到 redis.h 头文件的末尾。这样做的目的是让所有其他包含 redis.h 的源代码都可以使用我们的函数。
2. 将 double_metaphone.h 和 double_metaphone.c 复制到我们的 src 目录下。
3. 将 getMetaphone 和 setMetaphone 函数命令作为新命令添加到 redis.c 中的 redisCommandTable。
4. 编辑 double_mectaphone.c 并为 getMetaphone 创建一个占位符函数。
5. 对 getMetaphone 函数来说，我们将通过调用 t_string.c 代码文件中的标准 getGenericCommand 实现获取 Redis 键对应的字符串值。
6. 继续编辑 double_metaphone.c，并为 setMetaphone 函数创建一个占位符函数。

二元变音

二元变音是一种语音编码算法，它将一个字符串转换为主要编码(primary encoding)和次要编码(secondary encoding)。以下示例使用了两本书及其对应的作者：

Infinite - PRIMARY: ANFN SECONDARY: ANFN
Jest - PRIMARY: JST SECONDARY: AST

David - PRIMARY: TFT SECONDARY: TFT
Foster - PRIMARY: FSTR SECONDARY: FSTR
Wallace - PRIMARY: ALK SECONDARY: FLK

Pride - PRIMARY: PRT SECONDARY: PRT
and - PRIMARY: ANT SECONDARY: ANT
Prejudice - PRIMARY: PRJT SECONDARY: PRJT

Jane - PRIMARY: JN SECONDARY: AN
Austen- PRIMARY: ASTN SECONDARY: ASTN

在编译 Redis 之后，我们运行下列 redis-cli 会话时就会发现 GETDBLMEAPHN 和 GET 命令表现得完全一致：

```
127.0.0.1:6379> SET metaphone:1 "Star Trek"
OK
127.0.0.1:6379> GETDBLMETAPHN metaphone:1 "Star Trek"
127.0.0.1:6379> GET metaphone:1 "Star Trek"
```

我们添加到 `redis.h` 的 `getMetaphone` 函数的函数原型接收指向的 `redisClient` 指针并返回空。为了扩展我们最初仅仅调用 `getGenericCommand` 的实现,我们将添加新的功能来处理当 Redis 键找不到的情况。也就是说,我们将返回请求的键的变音。如果字符串不是数据库中现存的 Redis 键,我们就可以使用这种快捷方法生成字符串的双元变音。

虽然 `getMetaphone` 方法的这种新的行为方式我们是在产品环境中进行设计和实施的,但却非常适合用来展示如何自定义命令。我们了解 RESP 的相关知识,因而能够解析命令中的返回值。如果返回 `$-1`,我们将使用 Redis 键和输出代码字符串来调用 `DoubleMetaphone` 函数。该字符串将包含已经转换好的双元变音。



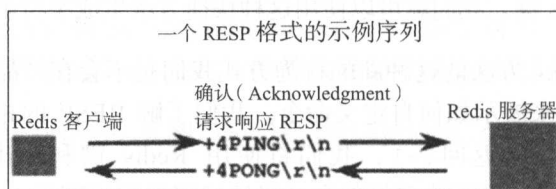
为了处理缺失 Redis 键的情况, `getMetaphone` 函数将返回该 Redis 键的双元变音:

```
void setMetaphone(redisClient *c) {
    c->argv[2] = tryObjectEncoding(c->argv[2]);
    setGenericCommand(c, 1, c->argv[1], c->argv[2], NULL, 0);
}
```

将来对双元变音命令的增强在于更改 `NewMetaString` 和 `DestroyMetaString` 及其他函数,从而可以使用 Redis 的简单动态字符串 (SDS)。SDS 定义在 `sds.h` 中,源代码在 `sds.c` 文件中。这些 sds 字符串结构在 Redis 代码库中使用得非常广泛,不仅用于支持 Redis 键和简单字符串,同时也支持例如列表、集合、有序集合、哈希及更多复杂值类型等 Redis 数据类型。通过重构变音代码来使用 Redis 的简单动态字符串,我们无须为变音命令实现自定义字符串类型,同时又充分利用了 Redis 代码库中现存的 sds 功能。

Redis 序列化协议

Redis 服务器与客户端之间的通信使用 Redis 协议规范，即 RESP。RESP 主要用于 Redis 通信，不过在 Salvatore San lippo 最近的项目 Disqu 中也采用了 RESP。在 Redis 官方文档中，RESP 被称为领域特定语言（DSL）。以这种视角来看待 RESP 是很有帮助的，特别是当我们讨论规范的细节时，它涉及了众多不同编程语言的不同客户端。RESP 能够序列化所有的 Redis 数据类型，可以轻松表示不同的数据类型，例如简单字符串、错误、整数、块字符串和数组。



RESP 客户端—服务器

虽然没有明确限定于 TCP 协议，但是 RESP 经常用于请求-响应模型的上下文中，以支持客户端-服务器应用程序。RESP 除了支持请求-响应模型之外，还支持：

- 流水线(Pipelining)，指的是客户端向服务器发送多条命令而无须等待每条命令的响应。

当服务器将通知发往拥有多个订阅方的信道时，发布/订阅模式会推送信道。当与 Redis 进行交互时，客户端将命令作为块字符串数组发送至服务器。Redis 服务器根据命令的类型使用 RESP 来解析，并按下列 RESP 类型进行响应。Redis 服务器响应分为以下 5 种类型，由第一个字节决定：

1. `+`代表简单字符串。
2. `-`代表错误字符串。
3. `:`代表整数字符串。
4. `$`代表块字符串。
5. `*`代表数组。

RESP 的终止字符串可以是回车或者是换行，用传统的 ASCII 表示为 `\r\n` (CRLF)。

对 RESP 简单字符串来说，开始字符串是加号标记字符`+`，之后跟着的字符串中的字符在结尾处不能包含回车或者换行，因为简单字符串是以 CRLF 结尾的：

```
+OK\r\n
```

当 Redis 服务器向发起请求的客户端发送简单字符串时，客户端应当获取响应中从初始的+到以\r\n结尾的字符。如果客户端发送了像 NOPE 这样无意义的命令，那么 RESP 响应将会如下所示：

```
-ERR unknown command 'NOPE'\r\n.
```

为了防范错误，Redis 服务器将返回 RESP 错误。它以负号字符“-”开始，之后跟着的是错误消息，并且也是以\r\n结束。当对数据类型进行了错误的操作或者将未知的命令发送给 Redis 服务器时，客户端在收到 RESP 错误时通常会抛出异常。

对于更具体的错误条件，例如在哈希键上执行 LPUSH 命令。这会导致 WRONGTYPE RESP 错误，随后是之前响应中的消息。

当客户端发起 INCR 和 INCRBY 命令时，服务器将返回 RESP 整数，如下是在 redis-cli 客户端上显示的：

```
127.0.0.1:6379> INCR global:counter
:1\r\n
127.0.0.1:6379> INCRBY global:counter 10
:11\r\n
```

RESP 整数类型以一个冒号字节开始“:”，之后是一个整数，同样以 CRLF 结尾。返回的整数字符串是一个有符号 64 位整数，随 Redis 客户端命令的不同而不同。

在 RESP 块字符串中，第一个字节是美元符号“\$”，其后是所包含字符串的整数长度，之后是 CRLF。接下来是真正的字符串数据。RESP 块字符串以第二个 CRLF 结尾。以下是一些 RESP 块字符串示例：

```
$-1\r\n
```

用于表示 null 值的 RESP 块字符串的长度为-1。

```
$0\r\n\r\n
```

空字符串的长度为 0，并以 2 个 CRLF 作为结束。

```
$25\r\nA world of handmade sound\r\n
```

之前的示例字符串“A world of handmade sound”，Redis 块字符串中编码后的数据长度

为 25，并且前后各有 1 个 CRLF。

客户端与 Redis 服务器之间使用 RESP 数组进行通信，并且 Redis 服务器返回的某些类型的响应也使用 RESP 数组。RESP 数组是其他 RESP 类型的集合，包括 RESP 简单字符串、整数和块字符串。RESP 数组的第一个字节是星号“*”，接下来的整数代表了元素的总数，并以 CRLF 结尾。RESP 数组中的每个元素都是一种 RESP 类型，并且它们之间无须相同。这导致了 RESP 数组可以拥有多种 RESP 类型的元素。RESP 数组也能由 RESP 数组元素组成，因而可以创建简单的数据层次或者树状结构。RESP 元素可以为 null。这可以通过使用块字符串 null 的语法 `$-1\r\n` 实现。

举例来说，包含两个整数 3 和 56 的 RESP 数组，看起来如下所示：

```
*2\r\n:3\r\n:56\r\n
```

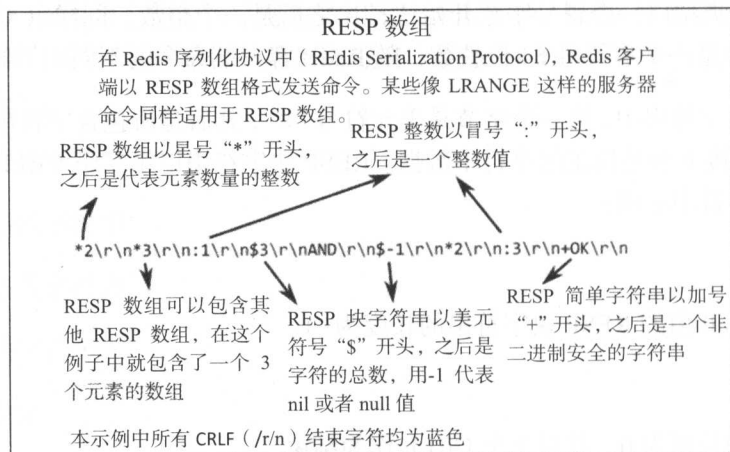
由 RESP 字符串、RESP 整数和 RESP 错误组成的混合 RESP 数组，如下所示：

```
*3\r\n$5\r\nhello\r\n:3\r\n-ERRWrong Type\r\n
```

最后，包含其他 RESP 数组的 RESP 数组如下所示：

```
*2\r\n*3\r\n:1\r\n$3\r\nAND\r\n$-1\r\n*2\r\n:3\r\n+OK\r\n
```

下图更好地展示了 RESP 数组和子类型：



RESP 数组

流水线

一种网络优化技巧是在 Redis 客户端和服务端之间使用流水线,使得客户端可以向服务器端发送多条命令而无须等待来自服务器的响应或者确认。Redis 中的流水线尝试减少服务器和客户端之间的往返时间(RTT),这是通过消除客户端为每个发送的命令接收服务器确认的需求达到的。在流水线中,客户端在发送多个命令之后立即解析服务器端响应。你可以通过 Netcat UNIX 程序直接使用流水线模式,也可以使用支持流水线的 Redis 客户端。

举个例子,可以利用流水线向运行中的 Redis 服务器发送多条命令,例如初始化 Redis 键模式。我们可以使用 Netcat 向运行中的 Redis 服务器发送以下命令,它将输入重定向到本地运行的 Redis 实例:

```
$ (printf "INCR\r\nglobal:book\r\nHMSET\r\nbook:1\r\ntitle\r\n\"Go Set a
Watchman\"\r\ncreator\r\n\"Harper Lee\"; sleep 1) | nc localhost 6379
$ (printf "INCR\r\nglobal:book\r\nHMSET\r\n"; sleep 1) | nc localhost 6379
+1
+ OK
```

我们从 Redis 服务器收到了单个响应+1 +OK。流水线方法为我们节省了 RTT 时间,对比之前我们从 redis-cli 上单独发送命令一共需要发送两次,而现在只需要 1 个 RTT 即可完成。

```
127.0.0.1:6379> INCR global:book
(integer) 1
127.0.0.1:6379> HMSET book:1 title "Go Set a Watchman" creator "Harper Lee"
OK
```

我们也可以在 Redis 客户端中使用流水线。使用 Python 推荐的客户端 redis-py 来进行演示(详情请参考 <https://github.com/andymccurdy/redis-py>)。

Redis RDB 格式

Redis 采用一种名为 RDB 的二进制格式来持久化内存数据快照。这是因为 Redis RDB 持久化默认模式允许内存中整个数据集在 Redis 意外退出时可以进行恢复,或者在 Redis 初始化时加载已存在的数据集。为了研究 RDB 文件格式,我们首先启动一个 Redis 实例,使用不同的数据结构存储一批不同的键,并使用十六进制编辑器和 Sripathi Krishnan 开发的 Redis RDB 工具(一个 Python 模块,详情请访问 <https://github.com/sripathikrishnan/redis-rdb-tools>)详细研究 RDB 的格式与结构。

RDB 格式不使用换行符或者空格作为分隔符。为了研究 Redis 数据结构的细节，我们将从 redis-cli 发送下列命令（在该示例中我们忽略了返回结果代码）：

```
127.0.0.1:6379> INCR global:book
127.0.0.1:6379> HSET book:1 author "Jane Austen"
127.0.0.1:6379> LPUSH book:1:edition:2 "copy 2" "copy 4"
127.0.0.1:6379> EXPIRE book:1:edition:2 400
```

在发送这些命令之后，我们来研究 dump.rdb。首先我们使用 vi 的十六进制模式打开该文件，然后依据下列表格中的每一行，对照并仔细检查选中的十六进制字符和对应的值：

十六进制	值	描述
52	R	每个 RDB 文件的头 5 个字节代表字符串 REDIS。这对于解析器来说很有用，可以确认这是否是一个二进制 dump 文件格式
45	E	
44	D	
49	I	
53	S	
30	0	接下来的字节 30 3030 36，意味着以大端格式表示的 RDB 版本号 006，即版本为 6
30	0	
30	0	
36	6	
fe	\xfe	fe 00 代表数据库选择器代码 fe 为 0
00	\x00	
00	\x00	00 是单个字节标志位，代表了值类型
0b	\x0b	
67	g	字符串键 global:book 以十六进制存储，同时值设置为 0
6c	l	
6f	o	
62	b	
61	a	
6c	l	
3a	:	
62	b	
6f	o	
6f	o	

续表

十六进制	值	描述
6b	k	
31	1	
10	\x10	<p>该字节表示存储在字符串中的值的编码类型为下列编码类型之一：</p> <ul style="list-style-type: none"> 0：字符串 1：列表编码 2：集合编码 3：有序集合编码 4：哈希编码 9：压缩哈希编码 10：压缩列表编码 11：整数集合编码 12：以压缩列表编码的有序集合 13：以压缩列表编码的哈希
62	b	<p>下一个存储在 dump.rdb 文件中的 Redis 键为 book:1:edition:2 list</p>
6f	o	
6f	o	
6b	k	
3a	:	
31	1	
3a	:	
65	e	
64	d	
69	i	
74	t	
69	i	
6f	o	
6e	n	
3a	:	
32	2	
00	\x00	接下来两个元素是 Redis 列表的内容，其中列表元素的大小为 6
06	\x06	
63	c	列表值“copy 4”以字符串形式存储
6f	o	
70	p	

续表

十六进制	值	描述
79	y	
20		
34	4	
08	\x08	08 是 0 的值类型，06 代表了下一个列表成员的长度
06	\x06	
63	c	列表值“copy 2”以字符串形式存储
6f	o	
70	p	
79	y	
20		
32	2	

使用 Redis 和 Python 创建协程

在印第安纳波利斯举办的 2015 Open Repositories 大会上，我遇到了 Mark Matienzo，他是美国数字公共图书馆的技术总监。他邀请我入队参加由大会主办的创意竞赛。我们的创意是基于缓存的关联数据片段服务器（Linked Data Fragments Server），它能帮助个人和组织提供简单并且易于理解的服务用来从图数据库中查询并返回 RDF 三元组，而不是去实现完整的、耗费资源的 SPARQL 终端接口，这种接口即便对大型网站来说也很难保持和运行。关联数据片段是由比利时根特大学的 Ruben Verborgh 首先提出的。它构造了由主语、谓语和宾语语句组成的三元组模式片段，用来查询关联数据库并返回匹配查询条件及元数据和分页信息的（由三元组构成的）关联数据片段。

虽然我们团队没有赢得创意大赛，但是来自 Mark 的鼓励及来自阿默斯特学院的 Aaron Coburn 的协助，促使我用 Python 和 Redis 启动了一个全新的开源项目，用来实现关联数据片段规范。源代码仓库在 <https://github.com/jermnelson/linked-data-fragments> 上。项目的第一个版本用在两个活跃项目 Islandora eBadge 和 bibcat.org 上。Islandora eBadge 项目允许组织发送 Mozilla Open Badges 给用户，同时将 Open Badges 结果存储到数字仓库中。另一个 bibcat.org 项目是我和国会图书馆约定的结果，为国会图书馆的新 BIBFRAME 关联数据词汇设计并实现基于关联数据的查询和展示系统。

对我来说，继续开发关联数据片段服务器的主要原因是对我所从事的数字图书馆系统

非常有用。关联数据片段服务器也能够帮助其他组织处理诸如国会图书馆和美国数字公共图书馆这样的大规模数据。另一个重要的原因是我对一个叫作 `asyncio` 的新的 Python 模块感兴趣。该模块支持异步 I/O, 提供全新的 (对于我来说) 网络编程模型。

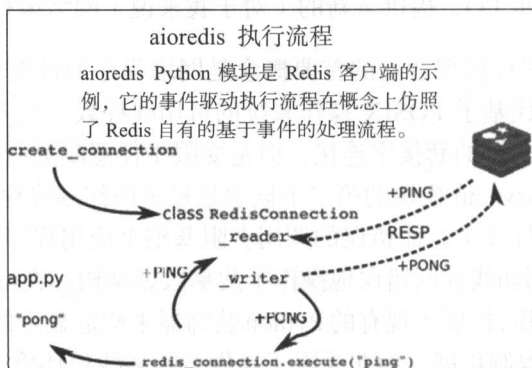
`asyncio` 模块和该编程模型背后的首要概念是网络程序的伸缩性应当受限于打开的套接字数, 而非受限于当代基于 POSIX 操作系统的可用线程数。大多数当代 POSIX 操作系统可以轻松处理上千个打开的套接字连接, 但是受限于任意时刻所能承受的线程总数, 通常范围在 100 个左右。`asyncio` 模块的第二个概念是显示协作多线程。在这里, 事件被立即处理或者等待更多的事件发生, 而错误的调用会阻塞整个应用程序的执行。这会导致许多微秒的多线程 BUG, 例如线程间错误地操作了共享数据结构。在 `asyncio` 模块中, Python 添加了额外的语法和操作, 扩展了现有的 Python 装饰器来构造显示的事件驱动的网络代码。这类代码能够基于 I/O 限制扩展, 而非受限于操作系统多线程环境中的可用线程数。

关联数据片段服务器包含了两个主要的文件 `server.py` 和 `cache.py`。`server.py` 实现了套接字服务器, 客户端会向该服务器发送主语-谓语-宾语三元组请求。我们使用一种简单的算法来解析这些请求, 生成三元组模式来查询 Redis 缓存。在 `cache.py` 中, 我们使用基于 `aioredis` 和基于 `asyncio` 的 Redis 客户端。这是用来与 Redis 缓存实例交互用的。`aioredis` 项目的 Git 仓库地址为 <https://github.com/aio-libs/aioredis/>。`aioredis` 的主要开发者是 Alexey Popravka。

`aioredis` 模块的主要 Python 模块 `__init__.py` 可从地址 https://github.com/aio-libs/aioredis/blob/master/aioredis/__init__.py 获取, 其中导入了该库中许多不同的组件, 包括 `connection.py`、`pool.py`、`util.py` 和 `errors.py`。Python 模块 `connection.py` 中定义了 `create_connection` 函数和 Python 装饰器 `asyncio.coroutine`。`create_connection` 函数中的第一个参数 `address` 可以是一个 Python 列表, 或者是元组, 代表主机和端口, 如果是字符串, 就代表运行时 Redis 服务器上的 UNIX 域套接字路径, 之后跟着的是 `db`、`password`、`encoding` 和 `loop` 等可选参数。

首先, `create_connection` 会从采用 `asyncio.open_connection` 打开的 Redis 连接来产生 Python `StreamReader` 和 `StreamWriter`。接下来, 会使用之前步骤中创建的 `reader` 和 `writer` 及其他 `create_connection` 参数来创建 `RedisConnection` 类的连接实例, `RedisConnection` 类提供一个 `_read_data` 协程以响应来自 Redis 服务器的输出, 以及两个重要的方法 `execute` 和 `execute_pubsub`。`hiredis` Python 模块用来在底层上解析 RESP 请求和响应, 它包装了 `Hiredis` 的 Redis C 客户端。`execute` 方法首先检测 `RedisConnection` 实例是否处于发布/订阅模式, 如果是的话, 那么在创建

`asyncio.Future` 之前返回错误信息。`execute_pubsub` 方法没有创建 `future` 对象，而是返回了 `asyncio.gather` 协程，用来协调发送和订阅信道及其关联模式。



`commands` 模块是由一组相关的命令组成的，这些命令用于哈希（`hash.py`）、列表（`list.py`）、字符串（`string.py`）、集合（`set.py`）、有序集合（`sorted_set.py`），以及其他数据结构的代码文件。`commands` 模块支持通用命令，对应 `generic.py` 文件；支持发布/订阅信道，对应 `pubsub.py` 文件；支持 Lua 脚本，对应 `scripting.py` 文件；以及支持 Redis 事务，对应 `transaction.py` 文件。

关联数据片段服务器使用 `aioredis` 模块来管理和响应那些遵循主语、谓语、宾语三元组格式的网络请求。虽然关联数据片段服务器也能够运行在除 HTTP 之外的其他协议上，但是服务器最初的设计遵循着使用类似 `http://linked-datastore-example.com/server?predicate=schema:creator&object=Mark+Twain` 这样的 URL 模式 HTTP GET 请求。在该 URL 示例中缺少了主语元素，该请求将搜索匹配马克吐温的 `schema:creator` 值的所有主语。之后，关联数据片段服务器将返回匹配该三元组片段的主语-宾语-三元组列表。

为了支持这种使用 `aioredis` 模块的设计，关联数据片段服务器实现了一种简单的 Redis 键模式。在我们的 RDF 实现中，我们假定三元组必须满足下列约束：

- 主语可以是空节点也可以是有效的 URL（URL 可以使用命名空间前缀）
- 谓语必须是有效的 URL
- 宾语可以是空节点、有效 URL 或者是字符串字面值

Redis 键模式为每个单独的主语、谓语或宾语采用了 sha1 哈希作为简单字符串键，用来存储序列化值。之后，我们将每个三元组键存储为以下格式——主语 sha1:谓语 sha1:

宾语 sha1, 对应的值存储为三元组的 JSON 格式关联数据。我们可以只简单地存储为 RDF XML、Turtle 或者其他 N 元组序列化格式。但是为了方便实现, 我们采用 JSON 格式, 因为它最常用。大多数三元组和三元组键都使用 TTL 命令设置了一周的有效期, 因而我们可以使用 Redis 的 LRU 缓存清除算法。

对于使用命名空间前缀的三元组来说, 我们在计算 sha1 哈希前扩展前缀以完善 URL。为了给键计算 sha1 哈希, 我们将 URL 和字面字符串规范化为单一标识符键。举例来说, 我们想要存储马克吐温的三本书: 《汤姆·索亚历险记》、《哈克贝利·费恩历险记》和《苦行记》的简单 RDF 图, 它们将以下列键进行存储。

三元组类型	值	sha1 哈希键
主语 URL	http://books.com/adventures-of-tom-sawyer	9192f6c2ea49440a15aa72c7d9c8c74f77ba2bf9
谓语 URL	http://schema.org/creator	e7c68409090a3d30933a819b3654b659c94cbc39
宾语字面值	Mark Twain	2b22164235bb360ad57c73ffffbd6550ddb366ef

在 Redis 数据库中, 我们会将三元组存储为一个键和一个整数。我们可以从 redis-cli 上重复上述 Redis 键模式:

```
127.0.0.1:6379> SET 9192f6c2ea49440a15aa72c7d9c8c74f77ba2bf9 http://
books.com/adventures-of-tom-sawyer
OK
127.0.0.1:6379> SET e7c68409090a3d30933a819b3654b659c94cbc39 http://
schema.org/creator
OK
127.0.0.1:6379> SET 2b22164235bb360ad57c73ffffbd6550ddb366ef "Mark Twain"
OK
127.0.0.1:6379> SET 9192f6c2ea49440a15aa72c7d9c8c74f77ba2bf9:e7c684090
90a3d30933a819b3654b659c94cbc39:2b22164235bb360ad57c73ffffbd6550ddb36
6ef '[\n {\n    "@id": "http://books.com/adventures-of-tom-sawyer",\n
"http://schema.org/creator": [\n        {\n            "@value": "Mark Twain"\n
}\n    ]\n }]\n'
```

在 cache.py 模块中我们可以看到, 当前的实现使用了 Redis 的 SCAN 命令来根据任意给定的三元组片段模式进行搜索。下面展示的是 get_triple coroutine:

```
@asyncio.coroutine
def get_triple(subject_key=None, predicate_key=None, object_key=None):
    redis = get_redis()
    pattern = str()
    for key in [subject_key, predicate_key, object_key]:
        if key is None:
            pattern += "*"
        else:
            pattern += "{}".format(key)
    pattern = pattern[:-1]
    yield from redis.scan(pattern)
    redis.close()
```

现在，我们可以使用该函数来匹配任何下列三元组搜索模式了：

使用 `schema:creator` 谓词和宾语字符串 Mark Twain 进行主语搜索：

```
*:7c68409090a3d30933a819b3654b659c94cb
```

使用 Node.js 和 Redis 实现 Todo 列表应用

node.js 版的 Todo 列表应用的作者 Amir Rajan 使用 Redis 实现简单的 Todo Web 应用程序。Todo 应用程序使用 node.js 及 `express` 和 `redis` 模块。我们首先克隆并安装应用程序所需的所有依赖，包括 Express 和 Redis 的 node.js 客户端。其中，Express 是一种快速的、极简主义的 node.js Web 框架。

```
$ git clone https://github.com/amirrajan/nodejs-todo.git
$ cd nodejs-todo
$ npm install
```

我们先在默认端口 6379 上启动 Redis 实例，然后从命令行上用 node.js 运行 `server.js`。

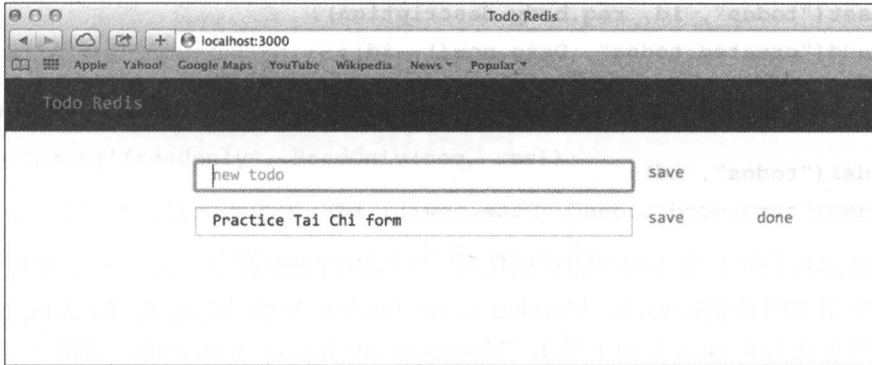
```
$ node server.js
```

现在，我们打开第二个终端窗口，运行 `redis-cli` 会话，检查 Redis 数据库的大小，然后运行 `MONITOR` 命令：

```
127.0.0.1:6379> DBSIZE
```

```
(integer) 1
127.0.0.1:6379> MONITOR
OK
```

然后,我们在 Web 浏览器里打开,提交我们的第一个 **Todo Redis**, 如下截屏所示:



Todo Redis 截屏

结果是 todos 哈希中设置了我们的第一条 Tai Chi 表单事件, 以 uuid 作为 Redis 键:

```
1436158403.745530 [0 127.0.0.1:61551] "hset" "todos" "c93e53d4-4284-43f1-
a3ef-d350ca805bbc" "Practice Tai Chi form"
```

如果我们关闭了 Redis 服务器,我们将在 node.js 应用程序中收到 `unhandledexception`:

```
events.js:72
    throw er; // Unhandled 'error' event
    ^
Error: Redis connection to 127.0.0.1:6379 failed - connect ECONNREFUSED
    at RedisClient.on_error (/Users/jeremynelson/2015/nodejs-todo/node_
modules/redis/index.js:189:24)
    at Socket.<anonymous> (/Users/jeremynelson/2015/nodejs-todo/node_
modules/redis/index.js:95:14)
    at Socket.EventEmitter.emit (events.js:95:17)
    at net.js:440:14
    at process._tickCallback (node.js:415:13)
```

Rajan 的 todo node.js 的功能有限。通过跟踪 todo 项的两个额外信息, 即创建时间, 以及完成或者取消时间, 我们就能给出任何 todo 项是何时创建、何时完成的。为了记录这

些信息同时提供额外的功能，我们将创建两个 Redis 有序集合：`created:todos` 和 `completed:todos`。在向有序集合中添加新成员时，我们将以 UNIX 时间戳作为分值，以 UUID 作为值。在 JavaScript 函数处理器的 `/todos/create` 路由中，我们在将 `todo` 项保存到最初的哈希值后，添加下列对 `redis` 模块的调用。

```
client.hset("todos", id, req.body.description);
client.zadd("created:todos", Date.now(), id);
```

然后我们将往 JavaScript 函数处理器的 `/todos/delete` 路由中添加类似的 `ZADD` 命令：

```
client.hdel("todos", id);
client.zadd("completed:todos", Date.now(), id);
```

为了测试我们针对应用程序的扩展代码，我们将首先保存 `server.js`，重启 `node`，然后往列表中添加两条新的记录，`Morning Code Review With Mike` 和 `Tai Chi 13 Posture Practice`。然后我们在 Web 界面上单击“Morning Code Review with Mike”旁的 `done` 连接。我们首先使用 `redis-cli` 来检查我们的两个新的有序集合是否会出现：

```
127.0.0.1:6379> KEYS *
```

- 1) "completed:todos"
- 2) "created:todos"
- 3) "todos"

然后，我们获取每个有序集合的值：

```
127.0.0.1:6379> ZRANGE created:todos 0 -1 WITHSCORES
```

- 1) "2b7dd99a-c1b3-4130-9372-00ea15f648f1"
- 2) "1436447038668"
- 3) "3a874008-7983-4861-b774-18baa5207fb3"
- 4) "1436447053529"

```
127.0.0.1:6379> ZRANGE completed:todos 0 -1 WITHSCORES
```

- 1) "2b7dd99a-c1b3-4130-9372-00ea15f648f1"
- 2) "1436447057307"

为了计算一条 `todo` 记录的总共完成时间，我们将获取“Morning Code Review with Mike item”记录的分值并相减 `1436447057307-1436447038668`。真难以置信，我们居然只花了 1.8639 秒就完成了代码审查！

复制与公共访问

现在,我们将为 todo 应用程序的数据存储添加冗余,创建一个从实例作为 todo 列表的备份。我们将在 Node.js server.js 文件中创建新的只读函数,使用 Redis 从实例来展示列表。首先,我们将创建一个 Express 路由,用来将 /readonly 映射到具体的函数:

```
function ReadOnly(req, res) {  
  res.render('readonly');  
}  
app.post('/readonly', ReadOnly(req, res));
```

下一步,我们将为从实例创建一份 redis.conf 的副本,并将其运行在端口 6380 上。我们在运行 server.js 时传入 readonly 参数,从而运行 ReadOnly 函数:

```
$ node server.js readonly
```

总结

本章首先深入研究了在多个 C 头文件和代码文件中 Redis 是如何处理并响应客户端请求的。我们研究了 processCommand 函数,该函数与 Redis 中的其他函数和操作都有关联。在之后的练习中,我们通过复制现有的 C 代码到开发 Redis 的分支中并进行改造,添加两个新的 Redis 命令。

第 5 章的主要内容是 Redis 序列化协议,以及非常详细的字节级别的 Redis 二进制存储格式。之后,我们展示了两个不同的项目,分别用到了 Python 和 Node.js 实现的 Redis 客户端。

现在你对 Redis 的内部工作机理有了更深入的理解。在第 5 章中,我们将展示如何向 Redis 中添加复杂的服务器端 Lua 程序,以替代自定义 Redis 命令。同时,我们会介绍一些流行的设计模式来使用 Redis 构建应用程序。最后会介绍如何在不同的 DevOps 环境中应用 Redis。

5

Redis 编程第二部分： Lua 脚本、管理与 DevOps

在本章中,我们首先着重讨论服务器端脚本 Lua 的功能与限制。Lua 脚本提供了为 Redis 添加复杂行为而无须修改其源代码的选择。我们将回顾之前章节中的示例程序,来看看这些应用程序采用 Lua 脚本能有怎样的改进与简化。之后,我们将转而讨论两个管理主题,即 Redis 主从复制与事务,是如何影响应用程序的设计的。我们将通过研究 Redis 在典型的 DevOps 环境中发挥的作用来结束本章。许多公司都采用 DevOps 改进信息与计算资源的交付。

在 Redis 中使用 Lua

Salvatore Sanfilippo 在他 2011 年的原创博客中提到了为 Redis 增加服务器端脚本的功能,他罗列了以下三点原因:

- 脚本能够通过减少服务器端和客户端之间的带宽来加快 Redis 在处理某些任务时的速度。典型的场景有为了读取值而发送多个单独的请求到 Redis,再到客户端对值进行计算,再将这些值回传给 Redis。
- Redis 中大多数工作流程受限于 I/O 而非 CPU,采用脚本将在两者间更好地取得平衡。
- 脚本允许 Redis 服务器代码库为了通用抽象而保持快速和精炼,同时给予用户添加特定服务器端功能的能力。



整篇博客可以在 <http://oldblog.antirez.com/post/redis-and-scripting.html> 上找到。

之后,服务器端脚本 Lua 编程语言的 2.6 版本被正式添加到 Redis 中。Lua 是一种快速且轻量级的编程语言,被设计用于内嵌到程序中,或者为其他编程语言添加脚本环境,特别是 C 和 C++ 程序及脚本游戏环境。Lua 在葡萄牙语中代表“月亮”,是由巴西里约热内卢的天主教大学的 LuaLab 发起并维护的。通过向运行中的 Redis 实例嵌入更复杂的逻辑和操作的方式, Lua 被用于扩展 Redis 服务器的功能。Lua 与其他脚本语言如 Python、Perl 和 Ruby 之间的差异在于,相较这些语言而言, Lua 非常快速和轻巧。Lua 采用 MIT 开源协议,源码可以从 <http://www.lua.org/> 上下载。

安装 Lua

需要根据操作系统来选择安装 Lua。对于 Linux 和其他派生于 POSIX 的操作系统,包括 Mac OS X,可以遵循 Lua 官网上的步骤进行安装:

```
curl -R -O http://www.lua.org/ftp/lua-5.3.1.tar.gz
tar xzf lua-5.3.1.tar.gz
cd lua-5.3.1
make {os-code} test
```



以上操作会下载 5.3.1 版本的 Lua,解压缩,然后根据操作系统进行构建。为了验证 Lua 成功编译并正确安装,将当前目录切换到 Lua 根目录,然后运行 test,结果如下所示:

```
$ make test
src/lua -v
Lua 5.3.1 Copyright (C) 1994-2015 Lua.org, PUC-Rio
```

在成功执行之前的 test 之后,你将在 Lua 交互模式下,通过调用 Lua 解释器学习和实验许多 Lua 的语法和类型。

```
$ lua-5.3.1/src/lua
Lua 5.3.1 Copyright (C) 1994-2015 Lua.org, PUC-Rio
>
```

虽然 Lua 脚本可以直接被 Lua 解释器调用并以独立程序的方式运行,但是当 Lua 内嵌在其他程序中时更有用。在 Redis 中, Lua 脚本运行在 Redis 服务器的事件循环中。这些脚本在从客户端加载之后运行,或者使用 EVAL 命令计算,或者通过使用 EVALSHA 命令,运

行 SHA1 编码的 Lua 脚本。EVAL 命令的语法如下所示：

```
EVAL lua_script number_of_keys key [key...] arg, [arg ...]
```

在上面的命令中, lua_script 参数是 Lua 5.1 脚本, 在 Redis 服务器的上下文中运行。number_of_keys 参数是之后参数的总数。它们代表的是 Redis 键。命令中的下一个部分是 Lua 脚本中使用的 Redis 键。最后会有 0 到多个额外的参数, 它们不是键, 是传递给 EVAL 命令的最后几个参数。

为了理解 Lua 语法的同时理解 Redis 对脚本语言的限制, 我们将从简单的示例开始, 再到稍加扩展的 Lua 脚本示例。

在 Python shell 上, 我们将引入 redis 模块, 创建 Python Redis 客户端, 然后创建 first_script Lua 脚本。该脚本返回 Hello Redis 字符串, 我们不传入任何键或者额外传递给 EVAL 命令的参数, 并将必填的键总数设置为 0:

```
>>> import redis
>>> datastore = redis.StrictRedis()
>>> first_script = """return "Hello Redis" """
>>> datastore.eval(, 0)
b'Hello Redis'
```

Lua 的核心语法故意保持得小巧。除了用于分隔名字和关键字的空格外, Lua 忽略其他空格和新行。标记 (或者其他语法元素) 之间的注释以 -- [[和]] -- 环绕。Lua 接受字母、数字和下画线的任意组合, 用作变量、表字段和标签的标识符, 不过变量名不能以数字开头。Lua 保留了 22 个关键字, 不能用于 Lua 函数中的名字。这些关键字包括控制流程关键字, 例如 if else、elseif、while 和 for, 以及定义函数关键字, 例如 goto、end 和 return。Lua 是大小写敏感的, 因此 addPerson 和 AddPerson 函数在 Lua 看来是两个不同的函数。另一个 Lua 变量名约定避免以下画线 “_” 开头。下画线开头后跟着大写字母约定为 Lua 环境变量, 例如版本 _VERSION。

使用 Redis Lua 脚本中的 print 关键字将在标准输出上打印内容。如果在 Lua 脚本中包含了打印语句的话, 将会在 Redis 的标准输出中展示内容。我们将看到下列 Lua 脚本, 开始是一段描述我们想要做什么的注释内容, 之后打印了从 Python shell 上运行的 Lua 程序的版本:

```
>>> second_script = """--[[ Prints Lua Version to Redis Output ]]--
print(_VERSION)"""
```

```
>>> datastore.eval(second_script, 0)
```

在该示例中,我们将在 Redis 运行的终端窗口中展示以下内容:

```
1139:M 27 Nov 12:03:49.640 * The server is now ready to accept
connections on port 6379
ions on port 6379
Lua 5.1
```

注意,我们的注释内容没有展示在 Redis 服务器的输出中,不过通过打印出全局变量 `_VERSION` 从而展示了内嵌在 Redis 中 Lua 的版本号 5.1。使用 Lua 的 `print` 语句,我们可以快速查看 Redis Lua 脚本的变量值。但是,在 Lua 脚本中,这不是用来调试错误的最好方法。从 Redis 3.2 开始,Redis 提供了集成的 Lua 调试器,我们将在本章稍后使用到。

Lua 变量是一级 (first-class) 的值,可以是以下八种不同类型之一: `number`、`string`、`boolean`、`function`、`nil`、`userdata`、`thread` 和 `table`。`number` 类型可以是整型或者浮点型,虽然能够以 32 位编译,不过 Lua 默认使用 64 位整型和 64 位双精度浮点。在我们的第三个 Lua 脚本中,你会发现 Lua `number` 既可以是整型值也可以是浮点数值,并且也会发现 Lua 是如何在这两者之间无缝转换的。下列 Lua 脚本首先创建了两个本地变量 `a` 和 `b`,并用 Lua 的 `type` 函数展示了 Lua 类型:

```
>>> third_script = ""--[[ Demos Lua int and float number type ]--
local a = 10
print(a)
print(type(a))
local b = a + 3.123
print(b)
print(type(b))""
>>> datastore.eval(third_script, 0)
```

从 Redis 服务器上可以看到如下输出:

```
10
number
13.123
Number
```

因为在 Redis 中 Lua 脚本是以严格模式运行的,所有的脚本变量需要本地化到脚本范围内。Redis 不允许在 Lua 脚本中使用全局变量。默认情况下, Lua 假设所有变量都是全局

的。由于 Redis 禁止使用全局变量，因此 Lua 中所有变量的声明必须使用 `local` 关键字。让我们看看下列 Lua 脚本片段能否运行，我们没有用 `local` 关键字修饰 `a` 变量：

```
>>> datastore.eval("""a = 10
print(a)""",0)
Traceback (most recent call last):
  File "<pyshell#48>", line 2, in <module>
    print(a)""",0)
  File "/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/
site-packages/redis/client.py", line 1899, in eval
    return self.execute_command('EVAL', script, numkeys,
        *keys_and_args)
  File
    "/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/
site-packages/redis/client.py", line 565, in execute_command
    return self.parse_response(connection, command_name, **options)
  File
    "/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/
site-packages/redis/client.py", line 577, in parse_response
    response = connection.read_response()
  File
    "/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/
site-packages/redis/connection.py", line 574, in read_response
    raise response
redis.exceptions.ResponseError: Error running script (call to f_928273b0
02b0116d3428bab44baa7c2af82dddf3): @enable_strict_lua:8: user_script:1:
Script attempted to create global variable 'a'
```

Lua 支持从字符串到数字的强制类型转换，数字可以从字符串转化而来，同时也有专门的函数将字符串转换为数字。在将参数传递给脚本时，这种值的转换非常有用。在 Lua 脚本中，传入的参数值是通过名为 `ARGV` 的表进行访问的。注意，传递给 Lua 脚本的所有 `ARGV` 参数值为字符串类型，因此如果你需要对值进行数值运算，就需要将它们转换或者强制转换为 Lua number 类型。以下是当我们在运行 `fourth_script` 时，我们将 1 作为可选参数传入时的代码：

```
>>> fourth_script = """--[[Demonstrates string-to-number using ARGV[1]
]]--
```

```

print(ARGV[1].." is a "..type(ARGV[1]))
local a = tonumber(ARGV[1])
print(a.." is a "..type(a))
print(ARGV[1] + 2)""
>>> datastore.eval(fourth_script, 0, 1)

```

上述代码在 Redis 服务器上的输出如下:

```

1 is a string
1 is a number
3

```

`fourth_script` 脚本引入了 “.” 字符串拼接操作, 允许你在脚本中将字符串拼接起来。Lua 的字符串类型是 8 位不可变字节序列, 并且是编码无关的。Boolean 和 nil 与其他编程语言类似, Boolean 的值为 true 或者为 false。nil 代表不存在的值, 以 boolean 值表示的话为 false 值。Lua 字符串可以以单引号 “'” 或者双引号 “” 进行分隔, 并且支持像 C 语言那样的转义序列, 例如换页符 \f, 换行符 \n, 回车 \r 或者纵向制表符 \v。Lua 的 8 位字符串可以包含任何以字面字符串表示的字节, 采用十六进制数值表示。我们可以在 Redis 中运行下列 Lua 脚本来演示这一点。

```

>>> datastore.eval("""return "\x48\x45\x4C\x4C\x4F\x20\x57\x4F\x52\x4C\x44" """, 0)
b'HELLO WORLD'

```

Lua 表是一种关联数组对象类型, 相比简单变量拥有更丰富的数据结构。Lua 表是一种通用聚集数据类型, 可以作为集合和列表使用。Lua 对这些不同的集合类型没有做区分, 不像其他编程语言拥有特定的列表和集合数据类型。Lua 表提供下标方式来存取成员值, 也可以使用字典那样的语法, 通过命名的键从表中获取特定的值。在 Redis Lua 脚本中, KEYS 和 ARGV 变量是 Lua 表类型, 存储了向 EVAL 和 EVALSHA 命令传递的 Redis 键和可选参数。需要注意的是, Lua 表不是从下标 0 而是从下标 1 开始的, 在第一次学习 Lua 表时会让人感到困惑。而且, 下标不能为 nil 或 NaN (not a number 的缩写)。

为了演示 KEYS 和 ARGV 是 Lua 表类型, 我们将运行 Lua 脚本来打印 KEYS 和 ARGV 变量的类型:

```

>>> datastore.eval("""print("KEYS type="..type(KEYS))
print("ARGV type="..type(ARGV))""", 0)

```

Redis 服务器上输出的打印内容如下：

```
KEYS type=table
```

```
ARGV type=table
```

我们也可以在 Redis Lua 脚本中使用花括号赋值给变量的方式创建新的表。在 `fifth_script` 中，我们将创建空表 `book`，将 `ARGV` 的第一个变量赋予 `bf:Title` 属性，然后返回 `bf:Title` 属性：

```
>>> fifth_script = ""--[[ Creates a Table for a Book based on ARGV ]--
local book = {}
book["bf:Title"] = ARGV[1]
return book["bf:Title"]""
>>> datastore.eval(fifth_script, 0, "Breakfast of Champions")
b'Breakfast of Champions'
```

当 Lua 表被当作数组使用时，可以依然沿用相同的基本表语法。在 Lua 中使用列表值创建数组十分简单。在 `sixth_script` 中，我们将定义一个本地工作周表，根据我们传递给 Redis Lua 脚本的值返回具体周几。注意，我们需要将 `ARGV[1]` 字符串转换为数值，以便使用基于下标的 Lua 表符号：

```
>>> sixth_script = ""--[[ Work Week Script takes number ARGV[1] and
returns day ]--
local work_week = {"Monday", "Tuesday", "Wednesday", "Thursday",
"Friday"}
return work_week[tonumber(ARGV[1])]""
>>> datastore.eval(sixth_script, 0, 2)
b'Tuesday'
>>> datastore.eval(sixth_script, 0, 0)
```

如果下标不存在，就会返回 `nil` 值，并且什么也没发生。将 4 作为 `ARGV[1]` 属性发送过去，可以从 `work_week` 表中得到期望的值：

```
>>> datastore.eval(sixth_script, 0, 4)
b'Thursday'
```

表在 Lua 中有不同的用法，花些时间学习表的细微差别会让你从中受益，并有助于增进你对在 Redis 服务器上运行 Lua 脚本的理解。更多有关 Lua 表的信息可以在 <http://lua-users.org/wiki/TablesTutorial> 找到。

函数类型允许 Lua 程序操作 Lua 和 C 函数。作为一等函数, Lua 函数是以引用而非值进行传递的, 因而内存使用上更高效。Lua 函数有下列语法:

```
functioncall ::= prefix_expression arguments
```

如果 `prefix_expression` 是 `function` 类型的话, 那么该函数 (处在 `prefix_expression` 的位置) 将以 `arguments` 参数被调用。如果不是函数类型, 那么 `prefix_expression_call` 元方法就会被调用, `prefix_expression` 会变成第一个参数, 而剩下的参数会成为原始参数。在 Lua 中定义函数, 首先需要以 `function` 关键字开头, 随后是在括号中间的参数, 还有函数体, 最后以 `end` 关键字结束。Lua 函数可以不使用 `return` 关键字返回结果。`seventh_script` 示例程序展示了从华氏温度转换为摄氏温度的 Lua 函数:

```
>>> seventh_script = ""--[[ Fahrenheit to Celsius Temperature Converter
]]--
local ftoC = function(f)
    return (f-32) * (5/9)
end
return ftoC(ARGV[1])""
```

现在, 我们可以使用一些示例华氏温度值运行 `seventh_script`:

```
>>> datastore.eval(seventh_script, 0, 95)
35
>>> datastore.eval(seventh_script, 0, 50)
10
>>> datastore.eval(seventh_script, 0, 32)
0
>>> datastore.eval(seventh_script, 0, 0)
-17
>>> datastore.eval(seventh_script, 0, -40)
-40
```

在本示例中本地 `ftoC` Lua 函数的调用语法与其他编程语言相似。在上述 `seventh_script` 示例中, `ftoC` 函数在被调用时使用的 `ARGV` 值, 就是我们在调用 `datastore.eval` 时传入的第三个值, 我们一共使用了 5 个不同的华氏温度。如果应用程序需要单位转换, 可以添加类似于此的华氏温度到摄氏温度的单位转换脚本, 以便在服务器端而不是客户端规范化数据。

不同于其他像 C 语言那样的编程语言，在 Lua 中函数可以被赋值给一个变量而成为匿名函数，无须事先为函数取名。Lua 函数和其他表达式一样被求值，函数被认为是一等的。就如我们在示例中看到的那样，我们在定义为函数的变量后面使用括号及 0 到多个参数。如果 Lua 函数拥有可变数量的参数，就需要使用省略号 "...”，并以右括号结束 “)”。具体的变量可以使用 select 关键字进行提取，或者通过省略号和花括号的方式存储到表中。Lua 函数可以返回任意数量、互不相同的单独的值，而不需要用容器对象包装后返回。eighth_script 首先定义了本地函数 OlympicMetals，返回三个字符串，代表不同的金属。

```
>>> eighth_script = """--[[ Lua function returning multiple variables
]]--
local olympicMetals = function()
    return "Gold", "Silver", "Bronze"
End
```

现在，我们用 Lua 脚本调用 olympicMetals，赋值三个变量并打印消息：

```
local gold, silver, bronze = olympicMetals()
print("1st="..gold.." 2nd="..silver.." 3rd="..bronze)"""
```

转换回 Redis 服务器输出，将看到如下代码：

```
1st=Gold 2nd=Silver 3rd=Bronze
```

将返回多个值的函数用括号包起来，会忽略除第一个之外的其他返回值。我们可以将 eighth_script 函数用括号括起来，然后观察它的返回值：

```
>>> eighth_script = eighth_script + "\nreturn (olympicMetals())"
>>> datastore.eval(eighth_script, 0)
b'Gold'
```

由于 Redis 只返回单个值，对返回多个值的 Lua 脚本进行求值就同让其只返回第一个值类似。在 Redis Lua 脚本中，我们有方法来应对该限制，使用花括号来包装任何返回多值的函数，以 Lua 表的形式返回。为了查看调用 olympicMetals 函数返回的内容，我们将修改 Lua 脚本，去除三个金属变量，取而代之的是将结果以 Lua 表的形式返回。在 Python Redis 客户端中，该返回值将以 Python 列表的形式返回下列三项内容：

```
>>> eighth_script = """--[[ Lua function returning multiple variables
]]--
```

```

local olympicMetals = function()
    return "Gold", "Silver", "Bronze"
end
return {olympicMetals()}""
>>> datastore.eval(eighth_script, 0)
[b'Gold', b'Silver', b'Bronze']

```

使用 Redis 的 KEYS 和 ARGV

我们已经了解了键和可选参数的用法。在 Redis 中,它们可以通过 Lua 表 KEYS 和 ARGV 进行访问。为了说明这一点,我们将运行 `ninth_script`, 该脚本会将变量 KEYS 和 ARGV 作为 Lua 表的成员回显到 Redis 客户端。

```

>>> ninth_script = ""--[[ Returns all KEYS and ARGV as members of a Lua
Table ]]--
return {KEYS[1], KEYS[2], ARGV[1], ARGV[2]}""
>>> keys_and_args = ["Airline:1", "Airline:2", "Singapore Airlines",
"Southwest"]
>>> datastore.eval(ninth_script, 2, *keys_and_args)
[b'Airline:1', b'Airline:2', b'Singapore Airlines', b'Southwest']

```

我们将重构这段脚本并命名为 `tenth_script`。我们不对键进行显示地访问,而是创建一个循环来迭代 KEYS 和 ARGV 中所有的值,并将结果以 Lua 表的形式返回:

```

>>> tenth_script = ""--[[ Demonstrates creating a Lua table with both
KEYS and ARGV ]]--
local airlines= {}
for i,k in ipairs(KEYS) do
    table.insert(airlines, k)
end
for i,k in ipairs(ARGV) do
    table.insert(airlines, k)
end
return airlines""
>>> datastore.eval(tenth_script, 2, *["Airline:1", "Airline:2",
"Singapore Airlines", "Southwest"])
[b'Airline:1', b'Airline:2', b'Singapore Airlines', b'Southwest']

```


在这段脚本中，`airlines` 变量前用 `local` 关键字进行修饰，并使用花括号赋值为空表。我们使用 `ipairs` 函数循环迭代 `KEYS` 和 `ARGV`，确保表是从 1 开始顺序访问的。这是基于位置访问的 `EVAL` 命令很重要的一个特性，它假设在 Lua 脚本中 `KEYS` 和 `ARGS` 是以顺序访问的。

在 Lua 脚本中可以使用 `redis.call` 或者 `redis.pcall` 函数调用 Redis 命令。这些函数将使用 Redis Lua 模块。这两个 Redis 函数之间的差异在于对错误的处理方式。在执行 `redis.call` 时产生的异常将被传递给发送 `EVAL` 命令的 Redis 客户端。另一方面，`redis.pcall` 将错误捕获到 Lua 表中，并将该表返回给客户端，这使得 Redis 客户端对错误的检测和处理更简单。如果使用的是 Redis 客户端，那么使用这两种方法对客户端造成的差异应该不大。例如，不管在 Lua 脚本中使用的是 `redis.call` 还是 `redis.pcall`，Python 的 Redis 模块都会产生 `redis.exceptions.ResponseError`。

由于 Lua 和 Redis 的数据类型并不相同，Redis 需要将数据转换为对应的 Lua 类型，并将脚本执行的结果转换为 Redis 的 RESP 值。下表展示的是 Lua 和 RESP 值之间对应的映射关系。

Redis 数据类型	Lua 数据类型
Redis 整型响应	数值 (Number)
Redis 块响应	字符串 (String)
Redis 多行块响应	表 (Table)
Redis 状态响应	含有单个 OK 字段的表
Redis 错误响应	含有单个 ERR 字段的表
Redis 空块响应	值为 False 的布尔值
Redis 整型 1 的响应	值为 True 的布尔值

我们在之前的章节中讨论过的 Redis 应用程序中的一种通用模式是维护一个全局计数器，并使用分隔符将其附加到键模式之后组成唯一键，然后将对应的数据和键插入 Redis 数据库中。举例来说，我们有一个全局的图书计数器，用来创建哈希键，并存放属性值。如果使用 Lua 脚本，我们可以将 Redis 的调用次数减半：

```
>>> add_book_lua = """local book_id = redis.pcall('INCR', 'global:book')
local book_key = "book: "..book_id
redis.pcall("HMSET", book_key, "title", ARGV[1], "author", ARGV[2])
return book_key"""
>>> datastore.eval(add_book_lua, 0, "Moby Dick", "Herman Melville")
b'book:1'
```

现在, 我们将从新的 Book 哈希中获取所有字段和值:

```
>>> for field, value in datastore.hgetall('book:1').items():
    print(field, value)
b'author' b'Herman Melville'
b'title' b'Moby Dick'
```

虽然可以通过使用 EVAL 命令来运行 Lua 脚本, 但推荐的方法是通过 Redis 进行脚本加载与执行。如果是小型 Lua 脚本, 那么在每次调用服务器时发送脚本所带来的开销小到可以接受。但是不利之处是在每次被应用程序调用时, 需要发送整个脚本文本。对于更复杂和大型的 Lua 脚本来说, 每次调用时在网络上发送完整的脚本文件让我们不得不考虑网络延迟。

在 Redis 中使用 Lua 脚本时的限制与考虑

阻塞: 注意那些长时间运行的 Lua 脚本或者又大又复杂的 Lua 脚本。由于 Redis 天生是单线程的, 这些脚本在执行完成之前会一直阻塞其他客户端!



脚本的原子性: 为了确保运行在任意 Redis 实例上的 Lua 脚本是纯函数。也就是说, 需要确信在从节点 1 和从节点 2 上运行的特定 Lua 脚本是一样的; 否则, 调试副作用及构造单元测试将变得更富挑战。任何 Redis 写命令在接收相同参数时应表现得完全一致。像 SPOP 这样随机返回一个值并写回 Redis 的命令是禁止使用的, 如果从 Lua 脚本中调用的话会导致错误。

受限的 Lua 库: Redis Lua 环境仅包含下列 Lua 库: table、string、math、debug、cjson 和 cmsgpack。

集群支持: Lua 脚本可以运行在 Redis 集群中, 前提是脚本操作的键处在同一个哈希槽上。

为了能够灵活地在 Redis 服务器上一次性加载 Lua 脚本, Redis 提供了 SCRIPT LOAD 命令。该命令接收 Lua 脚本字符串并返回该脚本的 SHA1 哈希值。然后客户端可以使用 EVALSHA 命令, 传入 KEYS 和 ARGV 参数, 调用 Lua 脚本的哈希摘要。SCRIPT LOAD 命令的语法如下所示:

```
SCRIPT LOAD script
```

Redis 对使用 SCRIPT LOAD 命令加载的 Lua 脚本进行 SHA1 计算, 返回可以用于

EVALSHA 命令执行的 SHA1 摘要。

除了能将 Lua 脚本加载到运行中的 Redis 实例外，Redis 还提供了其他三种 SCRIPT 子命令，用于帮助管理 Redis 服务器脚本缓存中的脚本：

SCRIPT EXISTS script_sha1 [script_sha1...]

SCRIPT EXISTS 命令接收一到多个 SHA1 摘要值，并检查该脚本是否存在于 Redis 服务器的脚本缓存中。检查的结果将以整数数组的形式返回，其中值为 1 代表该 SHA1 摘要存在，而 0 代表该 SHA1 摘要不存在于脚本缓存中。

SCRIPT KILL

SCRIPT KILL 命令专门用于长时间运行的 Lua 脚本，这些脚本可能存在内部错误或者挂起在一个竞争条件上。由于 Redis 服务器是单线程的并且会在每次 EVAL 或 EVALSHA 调用时阻塞，SCRIPT KILL 命令将中断执行进程，并返回一个错误给脚本和客户端。

SCRIPT FLUSH

最后一个命令是 SCRIPT FLUSH，它将清除 Redis 服务器的脚本缓存，同时任何之后的 EVALSHA 命令将返回错误，直到使用了 SCRIPT LOAD 命令加载了新的 Lua 脚本为止。

回到我们的 add_book_lua 脚本中，我们可以采用如下方式为该脚本产生 SHA1 摘要：

```
>>> add_book_sha1 = datastore.script_load(add_book_lua)
>>> add_book_sha1
'946ba456ead00a1787f6579097fc8df2fb30e17b'
```

有了该函数的 SHA1 值，我们就能在不同的客户端上使用该哈希摘要。在本示例中，特指我们的 redis-cli 客户端。我们将首先检测该脚本是否存在，然后通过调用 EVALSHA 命令添加新的图书哈希：

```
127.0.0.1:6379> SCRIPT EXISTS 946ba456ead00a1787f6579097fc8df2fb30e17b
1) (integer) 1
127.0.0.1:6379> EVALSHA 946ba456ead00a1787f6579097fc8df2fb30e17b 0 "I
Know Why the Caged Bird Sings" "Maya Angelou"
"book:2"
127.0.0.1:6379> HGETALL book:2
1) "title"
2) "I Know Why the Caged Bird Sings"
```

- 3) "author"
- 4) "Maya Angelou"

如果 Redis 脚本哈希被清空,那么我们会在尝试调用 Lua 脚本的 SHA1 摘要添加第三本书时收到如下错误:

```
127.0.0.1:6379> SCRIPT FLUSH
OK
127.0.0.1:6379> EVALSHA 18b5c2930c60be193478b990e2c8d5afda9116e4 0 "The
Adventures of Sherlock Holmes" "Sir Arthur Conan Doyle"
(error) NOSCRIPT No matching script. Please use EVAL.
```

Redis 中的高级 Lua 脚本

我们已经对 Redis 中的 Lua 脚本有了基本的理解,我们将重构之前章节中的一些示例程序来使用 Lua 脚本。我们也将注意到在使用 Lua 脚本时可能会发生一些潜在的问题,包括引入这些更改带来的隐藏的复杂性及对性能的影响。Lua 脚本是一种有价值的工具,但是必须在 Redis 服务器端脚本受限环境中使用。

在本书第 3 章中,我们将修改茶和咖啡的 Python 代码,使用 Lua 脚本做位图运算,最后在本书第 4 章中,我们修改了关联数据片段和 Node.js 应用程序来使用 Lua 脚本。

MARC21 数据提取

我们将使用加载了的 Lua 脚本实现第 1 章中的示例程序,将 MARC21 记录采集为 Redis 键模式结构。我们想要将处理流程转移到 Redis 服务器的主要原因是想减少客户端和服务端之间 Redis 命令的数量,取而代之的是单次调用 EVALSHA 命令,将我们想要的所有信息填充到数据库中的键。我们将从创建名为 `marc_ingestion_script` 的 Redis Lua 脚本开始:

```
>>> marc_ingestion_script = """--[[ MARC ingestion Lua script ]]-
local marc_key = KEYS[1]
if redis.call("exists", marc_key) < 1 then
    marc_key = "marc:"..redis.call("incr", "marc")
end
```

该 Lua 脚本首先为变量 `marc_key` 分配 `KEYS[1]`,然后检测 Redis 中是否存在

marc_key。如果 marc_key 不存在，就以 marc: 字符串拼接上从 Redis 调用全局 MARC 记录变量递增后所返回的整数。

```
local marc_fld_id = redis.call("incr", marc_key.."":"..KEYS[2])
local marc_fld_key = marc_key.."":"..KEYS[2].."":"..marc_fld_id
```

下一步，组合字段 marc_key 和 KEYS[2] 的值（我们假设该值为 MARC 字段代码）所得的全局变量递增之后用于创建 MARC 字段 ID。最后，我们将拼接原始 marc_key、MARC 字段代码和 MARC 字段 id。

```
redis.call("lpush", marc_fld_key)
```

然后，我们将 marc_fld_key 添加到列表 marc_key 中。该列表以倒序的方式存储所有添加到数据存储中的字段。

```
for i,k in ipairs(KEYS) do
  if i > 2 then
    redis.call("HSET", marc_fld_key, k, ARGV[i-2])
  end
end
return marc_fld_key""
```

因为其余的键是哈希 marc_fld_key 的字段，它们的顺序很重要，所以我们调用 ipairs 方法来确保对 KEYS 的迭代是从下标 1 开始的。由于 KEYS[1] 和 KEYS[2] 已经在使用了，我们将那些从 KEY[3] 开始的每个键映射到变量 ARGV 表对应的值。最后一行返回 MARC 字段的新 Redis 键。

在 Python shell 上，我们将执行 SCRIPT LOAD 命令返回的结果保存在 marc_ingestion_sha1 变量中留待之后使用：

```
>>> marc_ingest_sha1 = datastore.script_load(marc_ingestion_script)
>>> marc_ingest_sha1
'90eba74ace34ee70ad8705a9baab9e888c5c7740'
```

根据 David Foster Wallace 在 *Infinite Jest* 中的 MARC 记录，我们先定义一个 Python 列表来存储 MARC 100 字段的键和参数，然后使用 EVALSHA 函数调用 marc_ingestion_script 函数。我们将 marc_ingest_sha1、键的个数（3）及字段列表作为参数传入：

```
>>> field = [None, 100, "a", "Wallace, David Foster"]
>>> datastore.evalsha(marc_ingest_sha1, 3, *field)
```

```
b'marc:1:100:1'
```

从哈希 `marc:1:100:1` 中获取子字段 `'a'`, 如下所示:

```
>>> datastore.hget('marc:1:100:1', 'a')
```

```
b'Wallace, David Foster'
```

对于完整的 MARC 245 字段及其多个 MARC 子字段来说, 我们首先创建键和参数列表, 然后将 `marc:1` 作为 `KEYS[1]` 来调用 `EVALSHA`:

```
>>> field245 = ["marc:1", 245, "a", "b", "c", "Infinite Jest :", "a  
novel", "David Foster Wallace"]
```

```
>>> datastore.evalsha(marc_ingest_sha1, 5, *field245)
```

```
b'marc:1:245:1'
```

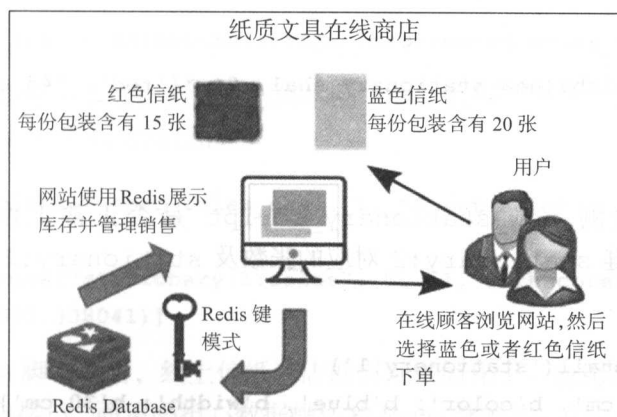
```
>>> datastore.lrange("marc:1", 0, -1)
```

```
[b'marc:1:245:1', b'marc:1:100:1']
```

`marc_ingestion_script` 让我们减少 Redis 客户端和服务端间的来回通信的次数, 从 3 到 5 次减少到单次 `EVALSHA` 调用。

纸质文具在线商店

在本书第 2 章中, 我们有一个纸质文具在线商店示例。在该示例中, 我们采用将逻辑相关的键关联起来的方式创建了 Redis 键模式, 用于管理两种类型的纸张库存, 当在线商店产生线上销售时进行更新。为了快速勾起你的回忆, 下图展示了来自本书第 2 章的示例。



为了将实现逻辑和键维护的功能从 Redis 客户端转移到 Redis 服务器端, 我们将创建两

个 Lua 脚本来处理 Redis 交互，同时减少我们调用 Redis 实例的次数。转移到 Lua 脚本的不利之处在于 Python 应用程序代码的可靠性降低了。我们现在需要程序员去理解 Lua 代码，而不是查看 Python 中个别 Redis 调用之间是如何相互联系的。另外一个希望将实现逻辑和键管理从 Python 代码转移到 Lua 的理由是项目考虑切换编程语言。新的应用程序将不需要复制逻辑代码，而只是使用合适的 Lua 脚本摘要和正确的 KEYS 和 ARGV 调用 EVALSHA 而已。

我们将重构纸质文具在线商店示例，创建一份名为 `new_stationary_script` 的 Lua 脚本，接收 0 个 Redis 键，5 个参数，分别代表信纸的颜色、宽度、高度，以及每份包装中含有的张数。该 Lua 脚本将返回新的信纸 Redis 键，如下所示：

```
>>> new_stationary_script = """local stationery_key = "stationery:"..
redis.call("incr", "global:stationery")
redis.call("hmset", stationery_key, "color", ARGV[1], "width", ARGV[2],
"height", ARGV[3])
redis.call("incrby", stationery_key..":sheets", ARGV[4])
redis.call("set", stationery_key..":inventory", ARGV[5])
return stationery_key"""
```

调用 `SCRIPT LOAD` 并将其 sha1 哈希值保存为变量 `new_stationary_shal`，之后就能使用该 Lua 脚本 sha1 哈希值调用 `EVALSHA`。当我们发送蓝色和红色信纸数据时，结果如下所示：

```
>>> new_stationary_shal = datastore.script_load(new_stationary_script)
>>> datastore.evalsha(new_stationary_shal, 0, *['blue', "30 cm", "40 cm",
20, 100])
b'stationery:1'
>>> datastore.evalsha(new_stationary_shal, 0, *['red', "45 cm", "45 cm",
15, 50])
b'stationery:2'
```

现在我们来检测 `new_stationary_script` 是否正常工作，分别获取哈希 `stationery:1`、键 `stationery:2` 对应的张数及 `stationery:2` 的库存，代码如下所示：

```
>>> datastore.hgetall('stationery:1')
{'b'height': b'40 cm', b'color': b'blue', b'width': b'30 cm'}
>>> datastore.get("stationery:2:sheets")
b'15'
```

```
>>> datastore.get("stationery:2:inventory")
b'50'
```

在使用该 Lua 脚本将这两种信纸添加到 Redis 之后, 下一步我们将创建第二个 Lua 脚本 `record_sale` 来记录在线销售:

```
>>> record_sales = """--[[ Records a Stationery Sale ]]--
local sales_sorted_set = KEYS[1]..":sales"
redis.call("decrby", KEYS[1]..":inventory", ARGV[1])
redis.call("zadd", sales_sorted_set, ARGV[2], ARGV[3])
return true"""
```

Lua 脚本 `record_sale` 接收信纸的 Redis 键作为 `KEYS[1]`, 包裹销售总数作为 `ARGV[1]`, UNIX 时间戳作为 `ARGV[2]`, 销售总金额作为 `ARGV[3]`。之后, 该脚本将库存 Redis 键减去销售的包裹数量。最后, `record_sale` 将新的记录添加到有序集合中, 该记录以 UNIX 时间戳作为分值, 以销售金额作为元素值。首先, 我们引入 `time` 模块生成 UNIX 时间戳, 并记录销售数据:

```
>>> import time
>>> first_sale = ['stationery:1', 2, time.time(), 20]
>>> first_sale
['stationery:1', 2, 1448837693.338041, 20]
```

在下面的代码片段中, 我们为内容为两个蓝色信纸包裹的首笔交易创建了一个 Python 列表:

```
>>> record_sales_shal = datastore.script_load(record_sales_script)
```

然后, 我们对 Lua 脚本进行测试, 将脚本的 SHA1 值保存为 `record_sale_shal`, 然后使用 `first_sale` 运行 EVALSHA:

```
>>> datastore.evalsha(record_sales_shal, 1, *first_sale)
1
>>> datastore.zrange("stationery:1:sales", 0, -1, withscores=True)
[(b'20', 1448837693.338041)]
```

有了这两份 Lua 脚本之后, 线上信纸商店的实现被简化了。应用程序开发者能够专注于网站特定的交互与接口, 而不必担心数据操作和 Redis 键模式。在创建一致的 Redis 键模式时, Redis 的 Lua 服务器端脚本的一个重要的作用是能够在应用程序的 Lua 脚本中处理绝

大多数的 Redis 键的创建与管理, 尽管大多数 Redis 对象映射器继续使用客户端代码来完成这些工作。

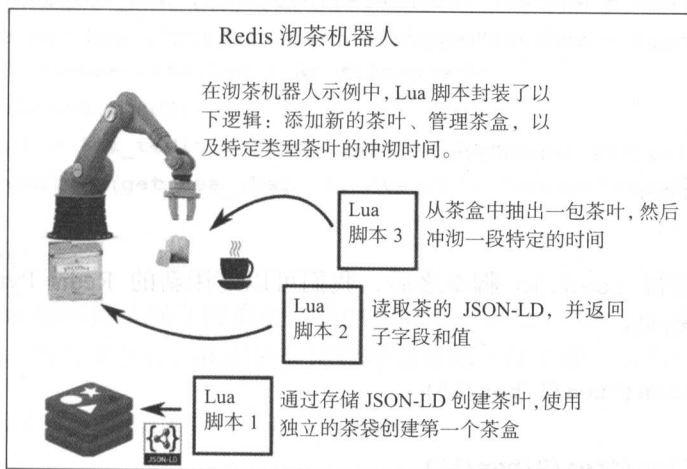
让 JSON-LD、Lua 和 Redis 协同工作

在本书第 3 章中, 我们使用 Redis 键模式完成了一个用于沏茶的 Web 服务, 其中使用正斜杠 “/” 作为键分隔符。运行在 Redis 中的 Lua 脚本能够直接使用更小的 JSON 结构, 例如关联数据。对于我们的沏茶示例来说, 我们将使用 <https://schema.org/> 和 <http://www.productontology.org/JSON-LD> 的词汇存储茶叶信息。为了避免维护将 JSON-LD 转换为 Redis 哈希的额外代码, 我们将为每种茶类型创建 JSON-LD 字符串。这样做有什么好处呢? 首先, 通过提供并使用存储为 JSON-LD 词汇的每种茶袋信息, 我们将使用相同的格式与服务于上下游客户需求的茶叶供应商进行互操作。下面展示的是按照 <http://schema.org/> 方案构建的茶冲泡服务中的 Earl Grey 茶的信息:

```
{
  "@context": {
    "": "https://schema.org",
    "pto": "http://www.productontology.org/id/"
  },
  "@id": "26bca550-db2b-445c-9253-4076e0bb968f",
  "@type": "Recipe",
  "cookTime": {
    "@type": "Duration",
    "@value": "PT5M"
  },
  "recipeIngredient": {
    "@type": "pto:Tea",
    "name": "Earl Grey"
  },
  "recipeInstructions": {
    "@type": "ItemList",
    "name": "Tea Box",
    "numberOfItems": 15
  }
}
```

我们创建的第一个 Lua 脚本通过将原始 JSON 数据存储到 Redis 数据库中递增的全局

变量中的方式, 添加一个新的茶品牌。同时, 也会创建茶袋并存储为 Redis 列表。JSON 在 Web 上的可用性意味着你可以使用 Redis 服务器上的 Lua JSON 库与 Redis 进行简单的互操作。



第一个 Lua 脚本将递增 `global/tea` 变量, 使用传入的 JSON 字符串和新的键创建并保存茶叶的键。最后, 我们使用用于 Redis Lua 脚本的 Lua 库 `cjson`, 为第一个茶盒创建一个列表, 并用茶袋进行填充:

```
local tea_key = "tea/"..redis.pcall("incr", "global/teas")
redis.pcall("set", tea_key, ARGV[1])
local box_id = redis.pcall("incr", "global/"..tea_key.."/box")
local box_key = tea_key.."/box/"..box_id
local box_json = cjson.decode(ARGV[1])
for i=1, box_json["recipeInstructions"]["numberOfItems"] do
    redis.pcall('rpush', box_key, i)
end
return tea_key
```

一旦 Lua 脚本被加载并将返回的 SHA1 摘要保存到变量中, 我们就能通过运行 Node.js 文件 `load.js` 中的 JavaScript 函数 `addAllTeas` 添加三个示例:

```
function addAllTeas(client) {
    console.log("Adding all Teas");
    fs.readFile("add-tea.lua", "utf8", function(err, data) {
        var add_tea_lua = data;
        client.script("load", add_tea_lua, function(err, data) {
```

```

    var add_tea_sha = data;
    var tea1 = addTea("earl-grey.json", add_tea_sha);
    var tea2 = addTea("lavender-mint.json", add_tea_sha);
    var tea3 = addTea("peppermint-punch.json", add_tea_sha);
  });
});
console.log("Finished");
return true;
}

```

在从命令行运行 load.js 脚本之后，我们可以使用新的 Redis Python 客户端实例 tea_redis 检验结果：

```

>>> tea_redis.llen("tea/1/box/1")
20
>>> tea_redis.llen("tea/2/box/1")
20
>>> tea_redis.llen("tea/3/box/1")
15

```

茶叶信息是以原生 JSON 字符串进行存储的，我们可以使用 Python JSON 模块在客户端对 JSON 字符串解码：

```

>>> import json
>>> earl_grey = json.loads(tea_redis.get("tea/3").decode())
>>> earl_grey["recipeIngredient"][0]
'Earl Grey'

```

虽然这个方案可以工作，但是我们还是能通过服务器端创建 Lua 脚本，用解码 JSON 并返回 JSON 哈希值的方式，改进我们的茶叶应用程序：

```

if redis.call("exists", KEYS[1]) == 1 then
    local raw_json = redis.call("get", KEYS[1])
    local tea = cjson.decode(raw_json)
    return tea[ARGV[1]][ARGV[2]]
else
    return nil
end

```

加载这段脚本,然后从 Python Shell 上调用它,使得我们能获取更小的原生 JSON 数据。如果 JSON 对象十分巨大,那么 Lua 脚本可能会花费过长时间解析并返回值,而 Redis 将会阻塞,直到会话结束:

```
>>> with open("get-tea-info.lua") as fileobject:
raw_lua = fileobject.read()
>>> get_tea_shal = tea_redis.script_load(raw_lua)
>>> tea_redis.evalsha(get_tea_shal, 1, "tea/1", "recipeIngredient",
"name")
b'Lavender Mint'
```

最后一份 Lua 脚本将从现存的茶盒列表中抽取第一袋茶,然后浸泡相应的时间,以此完成沏茶的过程。当到点之后,我们的应用程序会通知等待的那个人茶已经准备好了。

```
if redis.call("llen", KEYS[1]) < 1 then
    return nil
end
local tea_key = string.sub(KEYS[1], string.find(KEYS[1], "tea/%d+"))
local tea = cjson.decode(redis.call("get", tea_key))
local cook_time = tea["cookTime"]["@value"]
local brew_seconds = tonumber(string.sub(cook_time, string.find(cook_
time, "%d+"))) * 60
local bag_key = KEYS[1].."/bag/"..redis.call("lpop", KEYS[1])
redis.call("set", bag_key, 1)
redis.call("expire", bag_key, brew_seconds)
return bag_key
```

在我们的 Lua 脚本里,首先检测传入的 KEYS[1] 是否为空,如果为空则返回空。下一步,我们采用 Lua 标准字符串库,使用匹配函数 string.find 的 Lua 模式构造茶键。该值用于传递给函数 string.sub 以获取茶键。之后,我们解码存储在茶键中的 JSON 值,使用 string.sub 和 string.find 函数提取分钟数,并乘以 60 得到总共的秒数。下一步,从第二个键即茶盒键来构造 bag_key。我们使用键模式 tea/{tea-id}/box/{teabox-id}/bag/{teabag-id} 构造茶袋键。通过运行 LPOP 命令返回茶盒列表中的第一个元素来获取 tea-bag ID。然后,在返回茶袋键之前,我们基于本地变量 brew_time 设置超时时间。直到该键被驱逐出 Redis 实例之前,我们都可以通过客户端使用 ttl 命令进行查询。

为了使用 Redis 测试我们的 Lua 脚本,我们将回到 Python shell 上,打开 Lua 脚本:

```
>>> with open("brew-tea.lua") as file_object:
    brew_tea_lua = file_object.read()
>>> brew_tea_shal = tea_redis.script_load(brew_tea_lua)
```

在将 SHA1 保存至变量 `brew_tea_shal` 之后，我们将调用 `EVALSHA` 并保存返回的 Redis 键 `tea-bag`：

```
>>> tea_bag_key = tea_redis.evalsha(brew_tea_shal, 1, "tea/2/box/1")
>>> tea_bag_key
b'tea/2/box/1/bag/1'
```

我们将使用 `ttl` 命令在 Redis 实例上查询剩余时间：

```
>>> tea_redis.ttl(tea_bag_key)
159
>>> tea_redis.ttl(tea_bag_key)
59
>>> tea_redis.ttl(tea_bag_key)
-2
```

在茶袋冲泡完成之后，Redis 向客户端返回 -2。正如我们在之前章节中注意到的那样，采用客户端向服务器轮询的方式并不是应用程序的最优化设计。相反，使用 Redis 键过期时间通知功能，就能观测键模式，并让应用程序响应键 `tea_bag` 过期时产生的事件。在这种情况下，沏茶机器人会拿走茶袋。该示例演示了如何在 Redis 应用程序里直接使用 JSON 和在服务器端执行更复杂的工作流，例如沏茶。

Redis Lua 调试器

在 Redis 3.2 版本中最大的改进就是专门的 Lua 调试器，为运行在 Redis 服务器上的 Lua 脚本进行故障排除。Lua 调试器运行在由主 Redis 事件循环派生的服务器进程中。这意味着多个调试会话可以同时运行。默认情况下，在调试会话期间更改的任何数据会在会话结束时回滚。调试器有一个可选模式可以开启，以将调试会话期间的数据变更持久化到 Redis 服务器上（虽然该同步调试模式会将 Redis 服务器在整个调试会话期间阻塞）。Redis 的 Lua 调试器使用客户端-服务器端模式。在此模式下，Lua 调试器在远程服务器上运作，并将结果反馈给客户端。这是一种 `redis-cli` 客户端的特殊模式。

为了展示 Lua 调试器的使用方法，我们需要运行 Redis 3.2 服务器和 Redis-cli 客户端。下一步，我们将创建一份小巧的 Lua 脚本，接受一个或者多个 e-mail，将它们添加到聊天

室, 存储为 Redis 列表, 最后返回当前聊天室中一共有多少个参与者。该 Lua 脚本 `chatroom.lua` 可以从本书网站上下载, 如下所示:

```
01 --[[ Lua script for simple chatroom management ]]--
02 local chatroom_key = KEYS[1]
03 for i, email in ipairs(ARGV) do
04     redis.call("LPUSH", chatroom_key, email)
05 end
06 return redis.call("LLEN", chatroom_key)
```

我们将从 `redis-cli` 上调用脚本以开启 Redis Lua 调试器, 传入 `-ldb`、`-eval` 和文件 `chatroom.lua` 的路径, 还包括 `chatroom` 键、一个逗号及存储在 `ARGV` 中的三份 e-mail 地址:

```
~$ redis-3.2.0/src/redis-cli --ldb --eval chatroom.lua aikido-fan:456 ,
    "mu@aiki.com" "at@maf.info" "kc@chiaikido.io"
```

Lua debugging session started, please use:

`quit` -- End the session.

`restart` -- Restart the script in debug mode again.

`help` -- Show Lua script debugging commands.

```
* Stopped at 2, stop reason = step over
```

```
-> 2 local chatroom_key = KEYS[1]
```

现在 `redis-cli` 处于特殊的 Lua 调试模式, 不再响应除调试命令之外的 Redis 命令。Lua 调试器以步进模式开始, 调试器停止在了 Lua 代码的第一行, 即将本地 Lua 变量设置为 `KEYS[1]` 值。步进 (别名为单个 `s` 字符) 命令将执行 Lua 代码, 即 `chatroom.lua` 中的第二行:

```
lua debugger> step
```

```
* Stopped at 3, stop reason = step over
```

```
-> 3 for i, email in ipairs(ARGV) do
```

在执行过第二行后, 我们使用 `print` 命令 (使用 `p` 字符别名) 可以看到当前会话中所有有本地变量的值:

```
lua debugger> print
```

```
<value> chatroom_key = "aikido-fan:456"
```

从 Lua 循环的第三行开始，我们将迭代一次循环，然后再次发送 print 命令：

```
lua debugger> s
* Stopped at 4, stop reason = step over
-> 4 redis.call( "LPUSH" , chatroom_key, email)
lua debugger> print
<value> chatroom_key = "aikido-fan:456"
<value> (for generator) = "function@0x257ad70"
<value> (for state) = { "mu@aiki.com" ; "at@maf.info" ; "kc@chiaikido.io" }
<value> (for control) = 1
<value> i = 1
<value> email = "mu@aiki.com"
```

我们发送两次 step 命令（使用了别名 s）然后使用 print 命令展示变量的值。我们可以看到循环的值和当前用于迭代的 i 计数器变量为 1，以及当前 email 变量值为 mu@aiki.com。

使用 Lua 调试器添加断点很简单，只需使用 break 命令（别名为 b），指定 Lua 脚本行号即可。我们将在第 6 行添加断点，然后发送 continue（别名为 c）命令以完成循环的迭代：

```
lua debugger> b 6
5 end
#6 return redis.call("LLEN", chatroom_key)
lua debugger> c
* Stopped at 6, stop reason = break point
->#6 return redis.call("LLEN", chatroom_key)
```

你也可以单独在 Lua 脚本中通过添加 redis.breakpoint() 表达式的方式添加断点。Lua 调试器命令 redis（别名为 r）允许你在调试时向服务器发送 Redis 命令，我们可以使用 Redis 命令 LRANGE 查看键 aikido-fan:456 所对应的当前元素：

```
lua debugger> redis lrange aikido-fan:456 0 -1
<redis> lrange aikido-fan:456 0 -1
<reply> [ "kc@chiaikido.io" , " at@maf.info" , " mu@aiki.com" ]
```

我们可以看到聊天室中所有参与者的邮箱。另一个有用的 Lua 调试器命令是 trace（别名为 t），用来显示 Lua 调试器会话的当前回溯信息（backtrace），还有 list（别名为 l）命令，传入可选的行号参数，用于展示当前位置周围的源代码：

```
lua debugger> t
In top level:
->#6 return redis.call( "LLEN" , chatroom_key)
lua debugger> list
1 --[[ Lua script for simple chatroom management ]]-
2 local chatroom_key = KEYS[1]
3 for i, email in ipairs(ARGV) do
4 redis.call( "LPUSH" , chatroom_key, email)
5 end
->#6 return redis.call( "LLEN" , chatroom_key)
```

为了退出调试模式, 我们可以发送 `abort` (别名为 `a`) 或者 `continue` 命令来放弃 `redis-cli` 会话, 回到标准的 Redis 操作模式, 或者发送 `quit` 命令来完全退出会话:

```
lua debugger> abort

(error) ERR Error running script (call to f_2c7fbbfbe9fe0c7d7cfe6a4a70212
a3147560f7d): @user_script:6: script aborted for user request

(Lua debugging session ended -- dataset changes rolled back)

127.0.0.1:6379>
```

随着对 Redis 应用程序 Lua 脚本的开发与故障排除, 你会发现 Lua 调试器是一个特别有用的工具。

Redis 的编程与管理

当你为顾客和股东编写应用程序时, 有许多 Redis 功能和运作模式提供了额外的能力和限制。正如我们所见的那样, 虽然可以通过向应用程序添加 Lua 脚本的方式将更多的业务逻辑和数据操作转移到 Redis 实例中去, 但是当中也有重要的限制和取舍。在下一节里, 我们将探索与应用程序编程相关的 Redis 管理功能。首先, 我们先从 Redis 的主从实例复制解决方案开始, 然后是 Redis 的事务支持。我们从中可以看到这些功能是如何为应用程序提供机会的, 以及如何权衡这些对应用程序运作带来的影响。

主从复制

在 Redis 编程中需要考虑的一个基础管理主题是有关何时采用主从复制的决定。这会影响到 Redis 的稳定性、可伸缩性和性能,以及应用程序使用的 Redis 数据。在复制方案中,Redis 的主节点实例拥有和其他独立 Redis 从实例完全一样的数据。我们可以利用这一功能做很多事情,包括自动化备份,将支持写操作的主节点从其他从节点隔离开,响应来自用户需求的峰值。我们将快速回顾有关启动最简单的一对一的主从复制配置。

首先,我们打开终端,启动主节点和从节点:

```
$ screen ../redis/src/redis-server
$ Ctrl-a
$ screen ../redis/src/redis-server --dbfilename slave.rdb --port 6380
$ Ctrl-a
```

在上面的代码中使用了 screen 命令,我们首先启动了空的新 Redis 实例,运行在默认 Redis 端口 6379 上,使用 dump.rdb 文件作为快照。然后使用 Ctrl-a C 命令,我们创建了新的窗口并启动了第二个 Redis 实例作为我们的从节点,使用 slave.rdb 作为其 RDB 文件名,并运行在端口 6380 上,以避免和默认主节点运行的默认 Redis 端口造成冲突。最后,我们创建新的窗口运行 Python 命令。我们将创建两个 Redis 客户端,其中一个连接到主节点,而另一个是用于复制 first_master 的 Redis 从实例 first_slave:

```
>>> import redis
>>> first_master = redis.StrictRedis()
>>> first_slave = redis.StrictRedis(port=6380)
>>> first_master dbsize()
0
```

现在,我们将添加一些数据到新的主节点上。为了制造大一点的数据集,我们创建了一个 Lua 函数,使用传入的值作为迭代次数进行迭代,并简单地创建递增的 ID 字符串值:

```
>>> increment_lua = """for i=1,ARGV[1] do local key='test:'.i redis.
call('SET', key, 'value='..i) end"""
>>> increment_lua = """for i=1,ARGV[1] do
    local key='test:'.i
    redis.call('SET', key, 'value='..i)
end"""
>>> first_master.eval(increment_lua, 0, 100000)
```

我们使用命令来确认在主 Redis 数据存储中已经有了 10 万个键, 如下所示:

```
>>> first_master.dbsize()
100000
```

通过运行 MGET 命令, 我们抽取部分样板键, 确保它们的值是依据我们在 Lua 脚本中指定的那样进行设置的:

```
>>> for value in first_master.mget("test:1", "test:50000",
"test:100000"):
    print(value)
b'value=1'
b'value=50000'
b'value=100000'
```

现在, 我们回到 first_slave 并检测该 Redis 实例是否为空; 我们将发送 DBSIZE 命令, 然后发送 SLAVEOF 命令以便开始同步那个当前拥有十万个测试键的主节点:

```
.>>> first_slave.dbsize()
0
>>> first_slave.slaveof(host='localhost', port=6379)
True
>>> first_slave.dbsize()
100000
```

使用 *Ctrl+A* 循环切换活动窗口, 我们来观察运行中的主节点, 并看到一旦接收到来自从节点发来的 SYNC 命令, 一个后台进程就开始运行了:

```
1076:S 25 Jul 14:38:46.556 * SLAVE OF localhost:6379 enabled (user
request)
1076:S 25 Jul 14:38:46.705 * Connecting to MASTER localhost:6379
1076:S 25 Jul 14:38:46.706 * MASTER <-> SLAVE sync started
1076:S 25 Jul 14:38:46.706 * Non blocking connect for SYNC fired the
event.
1076:S 25 Jul 14:38:46.706 * Master replied to PING, replication can
continue...
1076:S 25 Jul 14:38:46.706 * Partial resynchronization not possible (no
cached master)
1076:S 25 Jul 14:38:46.706 * Full resync from master:
```

```
1b82a2315ec3c9daf5a37a7a11360469770494a7:1
1076:S 25 Jul 14:38:46.757 * MASTER <-> SLAVE sync: receiving 217808
bytes from master
1076:S 25 Jul 14:38:46.758 * MASTER <-> SLAVE sync: Flushing old data
1076:S 25 Jul 14:38:46.758 * MASTER <-> SLAVE sync: Loading DB in memory
1076:S 25 Jul 14:38:46.769 * MASTER <-> SLAVE sync: Finished with success
```

为了查看我们的从节点是否拥有了 Redis 主节点的内容，我们将确认 `first_master` 和 `first_slave` 是否拥有相同的值：

```
>>> for key in ["test:345", "test:67864"]:
print(first_master.get(key), first_slave.get(key))

b'value=345' b'value=345'
b'value=67864' b'value=67864'
```

Redis 的重新同步进程的近期改进是实现了 `PSYNC` 命令以提升持久性，同时减少了在标准 `SYNC` 命令执行期间的网络流量，即使主从节点间的复制连接在标准命令 `SYNC` 期间中断了。假如设置了 `repl-backlog-size` 指令并同时指定了大小，主节点会为复制流保存一份内存缓冲区（in-memory backlog）。即使主从节点之间的复制连接中断，缓冲区允许从客户端继续复制主节点的快照，以替代全量复制。该功能减少了网络流量并提升了稳定性，即使在 Redis 主从节点之间存在网络延时。

使用 MULTI 和 EXEC 实现事务

Redis 命令被置于 `MULTI` 和 `EXEC` 命令中间以单一顺序运行，由此构成了事务。Redis 的事务和基于 SQL 的关系型数据库的事务的差别在于无法获取值进行操作，这在 SQL 事务中却是可以的。更为重要的不同之处在于，如果执行命令时发生错误，对比 SQL 关系型数据库事务会进行回滚的方式，Redis 会继续执行剩余队列中的命令并且不会回滚已经执行成功的命令。

为了理解 Redis 事务是如何排队并从现存 Redis 实例 `first_master` 上顺序执行命令，我们将为 Redis 键 `movie-attendance` 创建一个事务，然后将在 `movie_attend_transaction` 中递增 `movie-attendance`：

```
>>> movie_attend_transaction = first_master.pipeline(transaction=True)
>>> movie_attend_transaction.incr("movie-attendance")
```

```
StrictPipeline<ConnectionPool<Connection<host=localhost,port=6379,db=0
```

在第二个终端窗口,我们将启动第二个 Python shell,并为 first_master2 创建一个 Redis 客户端,然后获取在第一个 Python shell 上递增的变量 movie-attendance:

```
>>> import redis
>>> first_master2 = redis.StrictRedis()
>>> first_master2.get("movie-attendance")
```

因为我们还没有执行事务,所以什么都没有发生。我们照如下方式在第二个 Python shell 上执行:

```
>>> first_master2.incr("movie-attendance")
1
```

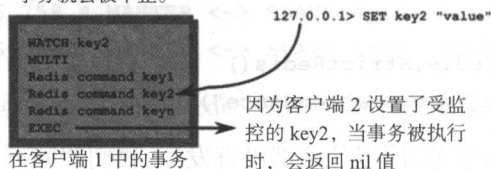
现在在我们最初的 Python 会话中,我们将对 movie_attend_transaction 发送 EXEC 命令。之后,我们将看到 first_master 上 movie-attendance 的值:

```
>>> movie_attend_transaction.execute()
[2]
>>> first_master2.get('movie-attendance')
b'2'
```

我们在客户端 first_master2 发送 INCR 命令之前就开启了事务,为什么 movie-attendance 的值为 2 呢? 这是因为 MULTI 命令是在 EXEC 命令被发送给事务时才执行的,而不是在 movie_attend_transaction 发送 INCR 命令时执行的。默认情况下,MULT/EXEC 创建了一个 Redis 命令队列。这些命令会在收到 EXEC 命令时执行。如果以值锁定的示例加以说明的话,那么可能会容易理解。因此,如果变量 movie-attendance 在事务中递增同时又被另一个客户端修改,也就像我们所演示的那样,那么该事务会由于加在 Redis 键 movie-attendance 上的乐观锁的缘故而失败。加锁的方式是使用检查并设置(CAS)方法的 WATCH 命令。

Redis 事务中采用 CAS 的乐观锁

使用 WATCH 命令的 Redis 事务展示了使用乐观锁的示例。如果另一个客户端更改了受到 WATCH 的键，那么事务就会被中止。



采用 CAS 的乐观锁

为了更好地演示乐观锁的使用，我们将使用 telnet 访问主 Redis 实例，并连同 MULTI 和 EXEC 一起运行 WATCH 来展示 Redis 事务的乐观锁。首先，我们回到 Python shell 上来并移除 movie-attendance 键：

```
>>> first_master.delete('movie-attendance')
1
```

现在我们使用 telnet 来连接，针对键 movie-attendance 使用 WATCH，然后用 MULTI-EXEC 运行下列事务：

```
bash-3.2$ telnet localhost 6379
Trying ::1...
Connected to localhost.
Escape character is '^]'.
WATCH movie-attendance
+OK
MULTI
+OK
SET movie-attendance 15
+QUEUED
EXEC
*1
+OK
```

现在我们从 Python shell 上检查 movie-attendance 的值：

```
>>> first_master.get('movie-attendance')
```

```
b'15'
```

事务正确执行, 并且如我们期望的那样, `movie-attendance` 的值被设置成了 15。现在我们使用 `WATCH` 重复会话, 看看会发生什么:

```
WATCH movie-attendance
+OK
MULTI
+OK
SET movie-attendance 23
+QUEUED
```

现在回到我们的 Python shell 上, 我们将修改 `movie-attendance` 的值然后回到 telnet 会话上运行 `EXEC`:

```
>>> first_master.set('movie-attendance', 5)
True
```

回到 telnet 会话上, 我们看到如下代码:

```
EXEC
*-1
```

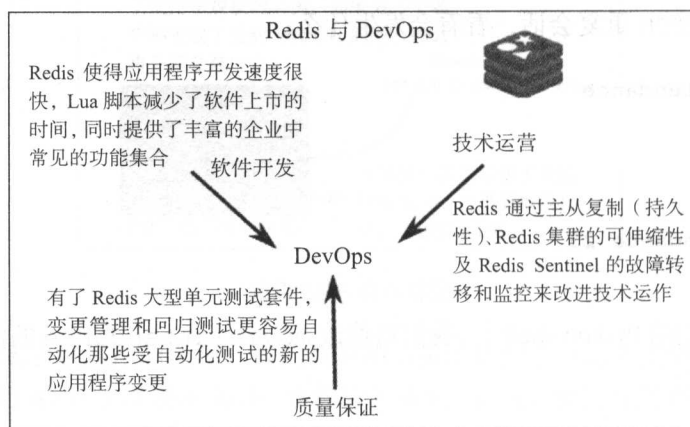
因为我们在 `movie-attendance` 上运行了 `WATCH`, 所以在 Python shell 上将值更改为 5 会导致我们在尝试运行事务后返回 -1。注意, 在 Redis 上运行的 Lua 脚本也是以相同的方式运作: 脚本中的操作是原子化的, 如果脚本运行失败, 那么作用在数据存储上的更改将会中止。

Redis 在 DevOps 中扮演的角色

DevOps 拥抱了企业技术的美好愿景: 将软件开发(工程化)、技术运营和质量保证这些传统孤立的部门有机组合起来, 并将这三种重要的功能合并成单一的组织单元。这个新的单元强调不同组件之间的通信与合作, 同时专注于自动化与集成开发、质量检测, 以及技术服务与资源的生产。通过聚焦在这些信息技术相关联的方面, DevOps 意味着减少新 IT 服务的上市时间或者发布时间, 更低的故障率与运行时 BUG, 同时缩短了产品服务灾难性故障的恢复时间。

之所以 Redis 能够很好地匹配该模型, 得益于其易于部署、对于核心及外延 Redis 技术

严格的单元及功能测试，以及通过诸如 Chief、Ansible、Puppet 和 Docker 等工具轻松实现自动化。



Salvatore Sanfilippo 撰写了有关将 Redis 用作“胶水”技术方面的内容，用于补充、完善传统数据库或者企业中的遗留系统。随着时间的推移，企业积累了一定级别的技术债务，造成对代码库进行修复或调整的大量工作。CAST 研究实验室估计每家企业中每 300,000 行代码的平均成本为 1,083,000 美元，换算成每一行代码的技术债务成本为 3.61 美元。在一个真正的 DevOps 环境中，运营代码例如 Lua 脚本为此开辟了新的道路。在这里，源代码可以逐渐积累并有助于可维护代码的总体负担。这些是联合运营需要解决的问题。

这并不是说可以完全避免引入新代码所带来的技术债务。由于基于敏捷流程会对代码库进行多轮迭代，因此 DevOps 能以积极响应的方式接纳并伴随着商业或者运营环境的变迁。造成这种变迁的原因是新的竞争者出现，或是企业在竞争环境下做出的决策改变。正常运作的 DevOps 的组织能较为理想地最小化代码的增加。这类代码是一些逐渐增长的“胶水”代码，用于直接连接那些不相容的供应商，例如独立应用程序、遗留系统，以及商业需求所需的任何当前运作的工作流。Redis 提供了另一种选择，允许对数据进行结构化来直面新的需求。这是一种激进的概念性的飞跃。同时，提供了一种类似于 Bridge 结构的设计模式，用来从实现中分离抽象概念，还提供了 Adapter/Wrapper/Translator 结构设计模式，用来将类的接口转换为另一种客户端或者进程期望的接口。

总结

本章首先详细研究了 Redis 服务器脚本语言 Lua，然后介绍了使用 Lua 脚本和 EVAL

命令的一些相关示例。之后讲到了使用 `SCRIPT LOAD` 和 `EVALSHA` 命令加载并执行 Lua 脚本。之后,我们对之前章节里的示例程序进行了回顾。这些示例包括了 MARC21 记录采集、在线信纸商店、关联数据片段服务器及众多 `node.js` 应用程序。同时,我们也看到了如何使用 Lua 脚本简化应用程序逻辑。第 6 章展示了不同的 Redis 运营模式,例如主从复制和事务是如何影响 Redis 应用程序设计的。

最后一节展示了在越来越流行的企业组织结构 DevOps 中,Redis 颇具灵活性,特别适合数据存储,并且满足了开发和运维员工提出的操作需求。

在本书第 6 章中,我们将更多地聚焦在 Redis 的运营方面。首先研究的是 Redis 集群的能力,它允许将大型数据带来的问题分散到多个 Redis 实例上。其次,我们会讲到有关 Redis Sentinel 的监控和故障转移的功能。

可伸缩性：Redis 集群和 Sentinel

本章首先介绍的是一种大型数据集可伸缩性的关键策略。该策略通过分区或者拆分的方式，将大型数据集分布到多个 Redis 实例上。不同的团队和项目已经采用多种算法对数据进行分区，其中包括 Redis 中最为成功的 Twitter 的 Twemproxy 项目。在过去的这些年，该项目为 Redis 描绘了背景和历史，促成了 Redis 最为重大的改变：Redis 第三版中将 Redis 集群包含在了稳定分支中。之后，我们将注意力集中在对 Redis 实例的分区及 Redis 集群采用的最终策略与实现方法上。然后，我们将对 Redis 集群进行实验，使用几个大型数据集，看看客户端应用程序代码需要如何修改以便使用 Redis 集群。

不管 Redis 为大型数据采用何种分区策略，对大量 Redis 实例的管理和支持促成了 Redis Sentinel 的开发与发布。它是一个包含在 Redis 内用于监控和故障转移的程序，解决了来自监控大量 Redis 实例的挑战，特别是在使用 Redis 的主从复制这一场景中。

数据分区的方法

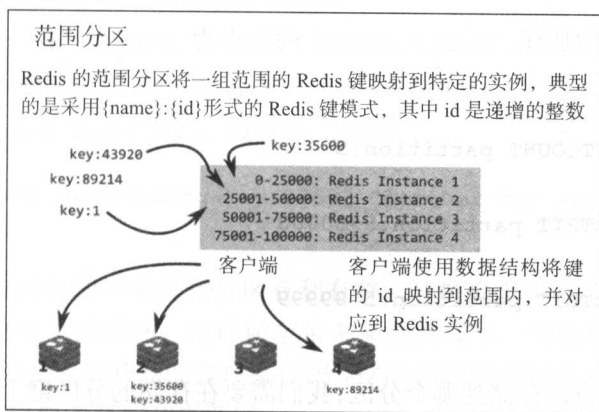
当单台机器的可用内存里无法装载大型数据库或者数据集时，数据分区就成为了一种重要策略，它将键进行分割并指派给特定的 Redis 实例。通过分区这一方式，单一 Redis 实例的计算能力和资源将不再成为限制。你可以对应用程序进行扩展延伸，以囊括多处理核心和多台机器，并通过网络接口、路由器和适配器连接到其他 Redis 实例。

在 Redis 中，通常有三种不同的方法来对数据进行分区：客户端分区、辅助代理分区，以及查询路由。在客户端分区中，分区逻辑包含在客户端代码中，它基于算法或者存储的额外信息或者兼而有之，用来选择正确的分区或者 Redis 节点。在辅助代理分区方案中，Redis 客户端连接到代理中间件，由它将客户端的请求路由到正确的 Redis 节点。在本章之

后的一个章节里,我们将介绍该方案中最为流行的 Twemproxy。最后一种 Redis 分区方法是查询路由:任一客户端查询集群中的随机节点将会被路由到包含键的正确节点上,这是 Redis 集群当前的实现方式。

范围分区

通常,对于范围分区来讲,不管是在服务器端还是在客户端,实现分区策略最简单的方法需要管理代码和数据结构来追踪哪些键被分派到哪个特定的实例上。范围分区的核心是依据传入的键是否落入特定的范围内,并将该键指派给该范围所对应的实例。



一个简化版的 Redis 范围分区是从明确数量的 Redis 实例开始。在此示例中,我们任意选取 5 个运行的 Redis 实例,并为每个实例指定一段 ID 范围。我们为每个实例存储一个 bitstring,并将对应范围 ID 全部置成 1。当使用全局递增生成一个新的键时,我们将使用 GETBIT 方法逐个检查每个分区,查看该键是否落在该分区范围内。一经确认,就将该键存储到分区所对应的 Redis 实例中去。

以下是一个简短的 Python 函数 set_range_partitions,它实现了为分区设置位的功能:

```
def set_range_partitions(datastore, partitions=5, size=20000):
    for i in range(0, partitions):
        key = "partition:{}".format(i+1)
        start = i*size + 1
        end = start + size
        for offset in range(start, end):
            datastore.setbit(key, offset, 1)
```

现在，我们有了 5 个分区键，存储了对应范围的 bitstrings。由于这些键并不消耗多少内存，我们可以将其副本分别存储于 5 个 Redis 实例中。首先，我们从 redis-cli 会话中检查 partition:1。

```
127.0.0.1:6379> BITCOUNT partition:1
(integer) 20000
127.0.0.1:6379> GETBIT partition:1 1
(integer) 1
127.0.0.1:6379> GETBIT partition:1 20001
(integer) 0
```

就如我们所期望的那样，partition:1 的大小为 20000，第一个设置的位始于偏移处 1，如下所示：

```
127.0.0.1:6379> BITCOUNT partition:5
(integer) 20000
127.0.0.1:6379> GETBIT partition:5 80000
(integer) 1
127.0.0.1:6379> GETBIT partition:5 99999
(integer) 1
```

为了计算新的键应该存储到哪个分区，我们需要在存储的分区键上执行 bitstring 操作。最简单的方法是发送一个 GETBIT 命令并以新的键的数字 ID 作为参数，检查每个分区的返回结果是否为 1：

```
127.0.0.1:6379> GETBIT partition:1 568
(integer) 1
127.0.0.1:6379> GETBIT partition:2 568
(integer) 0
127.0.0.1:6379> GETBIT partition:3 568
(integer) 0
127.0.0.1:6379> GETBIT partition:4 568
(integer) 0
127.0.0.1:6379> GETBIT partition:5 568
(integer) 0
```

于是, 我们使用 `partition:1` 来存放键 `interesting-key:568`:

```
127.0.0.1:6379> SET interesting-key:568 "Some data"
```

```
OK
```

对于第二个 ID 位 83697 的键来说, 我们将重复之前检测每个分区 `bitstring` 的流程(这里省略了前三个 `GETBIT` 的检查结果):

```
127.0.0.1:6379> GETBIT partition:4 83687
```

```
(integer) 0
```

```
127.0.0.1:6379> GETBIT partition:5 83687
```

```
(integer) 1
```

第二个键存储在集群中第五个 Redis 实例中, 该实例运行在 6382 端口上。我们新建一个 `redis-cli` 会话并连接上 Redis 实例, 设置第二个键:

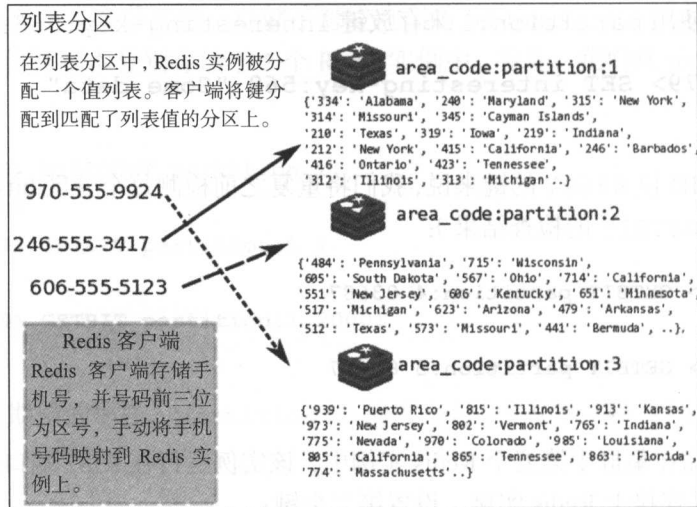
```
127.0.0.1:6382> SET interesting-key:83697 "Another key with info"
```

```
OK
```

使用范围分区方法对数据集进行分区有利有弊。从概念上讲, 范围分区最容易理解和实现。但是, 正如你所见的那样, 这样做会带来额外的成本, 包括用于追踪分区的 Redis `bitstring` 数据结构, 以及定制开发的客户端代码。这些客户端代码用于管理将键分配到分区及从运行中的 Redis 集群中获取和更新键。

列表分区

与范围分区相似, 列表分区指的是为分区指定一个列表值, 如果 Redis 键拥有列表值中的一个, 那么就将该键分配到那个分区。为了展示列表分区, 我们将以简单的电话应用为例, 将来自美国全国各地的电话号码存入三个运行中的 Redis 实例。根据每个 Redis 实例分配的区号决定键该分配给哪个实例。



就像范围分区那样, 使用列表分区同样需要中间数据结构, 以支持数据存储中键的分配与追踪。在此案例中, 我们将填充三个哈希, 每个哈希中的字段名为区域代码, 对应的值为地理区域名称 (国家或者州)。因为区域代码在数字上不是连续的, 所以我们无法使用范围分区。

首先, 我们打开一份以制表符分隔的区域代码文件, 从每一行中抽取区域代码和地理名称, 然后将前 106 个区域代码分配到第一个分区, 接下来的 106 个区域代码分配到第二个分区, 最后, 将最后 107 个区域代码分配到第三个分区。我们将三个 Redis 键以 `area_code:partition:{id}` 的形式存储, 同时第一个 Redis 实例将以第一个分区运作:

```
def assign_codes_to_partitions(filename, datastore):
    with open(filename) as area_codes_file:
        area_codes = area_codes_file.readlines()
        area_code_shard1 = "area_code:partition:1"
        area_code_shard2 = "area_code:partition:2"
        area_code_shard3 = "area_code:partition:3"
        for i, row in enumerate(area_codes):
            fields = row.split("\t")
            code = fields[0]
            geo_name = fields[1]
            if i < 106:
                slot = area_code_shard1
            elif i >= 106 and i < 212:
```

```

        slot = area_code_ shard 2
    else:
        slot = area_code_ shard 3
    datastore.hset(hash_key, code, geo_name.strip())

```

为了确保第一个 Redis 节点确实存储了三个键并且每个哈希表的大小的确和我们预期的那样, 我们通过 redis-cli 会话来验证这一点:

```

127.0.0.1:6379> DBSIZE
(integer) 3
127.0.0.1:6379> HLEN area_code:partition:1
(integer) 106
127.0.0.1:6379> HLEN area_code:partition:2
(integer) 106
127.0.0.1:6379> HLEN area_code:partition:3
(integer) 107

```

我们需要另一个函数, 接收电话号码、值列表(姓名、地址、电话或者座机)及集群节点列表作为参数, 根据区域代码查找到对应的节点, 并将手机号码存储到 Redis 实例节点上:

```

def save_phone_number(phone, values, cluster):
    area_code = phone[0:3]
    if cluster[0].hexists("area_code:partition:1", area_code):
        shard = cluster[0]
    elif cluster[0].hexists("area_code:partition:2", area_code):
        shard = cluster[1]
    else:
        shard = cluster[2]
    shard.hmset(phone, values)

```

我们将从交互式 Python shell 上测试第一个电话号码, 执行 Python 函数 save_phone_number, 如下所示:

```

>>> save_phone_number(
    "719 555 1212",
    {"name": "Jeremy Nelson",
     "type": "Mobile"},
    cluster)

```

基于列表分区的原理, 电话号码"719 555 1212"以哈希表的形式存储在集群中的第三个节点上:

```
127.0.0.1:6379> HEXISTS area_code:partition:2 719
(integer) 1
```

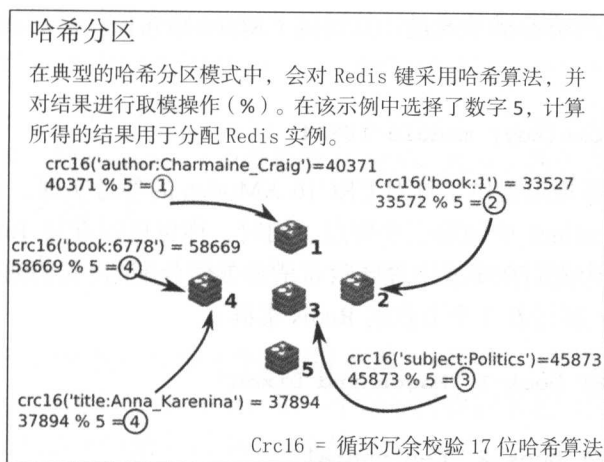
我们可以打开另一个终端窗口, 使用 `redis-cli` 连接上第三个节点, 获取"719 555 1212"键的所有字段和值:

```
$ ~/redis/redis-cli -p 6381
127.0.0.1:6381> hgetall "719 555 1212"
1) "type"
2) "Mobile"
3) "name"
4) "Jeremy Nelson"
```

经过一番设置, 我们应该能够将所有这些手机号码分布到三个分区中去。我们的分区策略中欠缺考虑的是随着数据集的增长添加额外分区的方法。由于我们的算法是依据区域代码分布的, 如果不重构列表, 我们是无法添加更多节点的。我们的策略也假设北美电话号码均匀分布在这些区域代码中。新的区域代码也会由北美电话区号规划管理 (NAPA) 在适当时机添加进来。虽然并非不可能, 但是我们可以手工实现一个重新分区的算法, 以便在 NANPA 添加新的区域代码时, 数量相等的区域代码会从各自的节点移出并重新分配到新的节点。不同于范围分区, 虽然不需要区域代码是顺序的或者是连续的, 但还是少不了要添加自定义客户端分区代码来管理小型 Redis 集群。

哈希分区

在哈希分区中, 哈希算法根据键计算出用于分区的哈希值。典型的 `hash` 函数根据键计算出值, 并基于数据存储中分区数量或者可用实例数对该值进行取模。在 2011 年的一篇名为 *Redis Presharding* 的博客中, Salvatore Sanfilippo 提出了一个基本的、简单的哈希算法, 该算法接收 Redis 键作为参数, 使用诸如 SHA1 或者 CRC 计算哈希值, 然后再取模以计算出键具体存放的位置或者节点。Sanfilippo 鼓励大家在搭建 Redis 集群时运行许多不同的实例。在他的例子中, 用于在客户端计算 Redis 键的哈希值的 Redis 实例共有 128 个。



Java 编程语言中广泛采用了哈希算法, 类中的方法根据类实例存储的数据进行摘要算法, 创建单个 32 位有符号哈希值。在使用 Java 客户端和 Redis 进行哈希分区的例子中, 由邮件地址构成的键通过调用 Java 的方法计算 e-mail 字符串的哈希值, 然后被路由到集群中的 Redis 实例, 并存储在 *email bucket* 中。

复合分区

在复合分区策略中, 键会通过范围分区、列表分区或者哈希分区之间不同组合的分区计算方式, 最后被分配到指定的实例中去。Redis 集群使用一种称作**一致性哈希**的复合分区形式。这种分区方式组合了哈希分区和列表分区的特征来计算键的归属实例。键的 CRC16 哈希值被称为哈希槽, 然后使用 16384 进行取或者 CRC16 模运算。Redis 集群用来计算键的哈希槽的具体算法是循环冗余校验 (CRC), 选用 17 位长度的多项式即 CRC16。理论上集群节点最多有 16384 个, 每个节点都运行着 Redis 实例。为了能更高效地使用 Redis 集群的一致性哈希算法, 集群中需要确保至少有三台 Redis 节点。

对于三个节点的 Redis 集群来说, 哈希槽的分配方式如下所示。

- 第一个主节点拥有 0~5500 哈希槽
- 第二个主节点拥有 5501~11000 哈希槽
- 第三个主节点拥有剩余的 11001~16384 哈希槽

一个键真正对应的哈希槽是通过计算键的 CRC16 哈希值，然后对 16384 进行取模得到的，如下所示：

```
HASH_SLOT = CRC16(key) modulo 16384
```

官方的 Redis 集群规范提供了针对 CRC16 XModem 的参考实现，请参考附录中可伸缩性：Redis 集群和 Sentinel 中的第三个要点。同时，你也可以查阅 Redis 源代码目录下的 `crc16.c` 代码文件。让我们在实战中理解哈希槽是如何分配的。我们通过 `redis-cli` 客户端，携带 `-c` 参数，连接上运行着 3 个节点的 Redis 集群：

```
127.0.0.1:9001> SET book:1 "Mason and Dixon"
OK
127.0.0.1:9001> SET book:2 "Centennial"
-> Redirected to slot [12948] located at 127.0.0.1:9003
OK
```

第一个键 `book:1` 的哈希槽通过计算 `crc16("book:1")` 并对 16384 取模的结果是 759。在这个特定的集群中，主节点常驻在端口 9001，并且被分配了从 0 到 5500 的哈希槽，因而客户端发起的 `SET` 命令就在原节点上执行。第二个键 `book:2`，该键的哈希槽通过计算 `crc16("book:2")` 并对 16384 取模的结果是 12938，将被分配到常驻端口 9003 的主节点上。Redis 提供了方便的 `CLUSTER KEYSLOT` 命令来执行哈希槽的计算，用来代替手工计算。

```
127.0.0.1:9001> CLUSTER KEYSLOT book:1
(integer) 759
127.0.0.1:9001> CLUSTER KEYSLOT book:2
(integer) 12948
```

键哈希标签

之前，我们讨论 Redis 集群中标准哈希槽分配的一个重要例外是在 Redis 键中使用哈希标签。它受限于用于计算哈希槽的哈希标签中的字符。在 Redis 键中，哈希标签指的是第一次出现左大括号和右大括号之间所包含的字符。这样做强制键能够驻留在集群中同一个哈希槽和 Redis 节点中。这一点很重要，因为 Redis 集群仅为多键（multi-key）命令提供有限的支持，同时仍能在整个集群范围内完整支持核心的 Redis 命令。如果需要使用多键命令，那么所有的键必须驻留在同一个节点，因此哈希槽计算就只受限于哈希标签。回到 `redis-cli` 会话中，我们先尝试在不使用哈希标签的情况下发送 `MSET` 命令：

```
127.0.0.1:9001> MSET book:3 "Shogun" book:4 "Gone Fishin"
(error) CROSSSLOT Keys in request don't hash to the same slot
```

这是因为键 `book:3` 的哈希槽通过 `crc16("book:3")` 计算并对 16384 取模后的结果是 8885。而经过计算, 键 `"book:4"` 的哈希槽为 4690。现在, 我们尝试相同的命令, 不过这次使用哈希标签 `{book}`:

```
127.0.0.1:9001> MSET {book}:3 "Shogun" {book}:4 "Gone Fishin"
OK
```

在该示例中, 我们期望两本书的键能够驻留在相同的同一节点上, 因此我们创建了 `{book}` 哈希标签并发送两个 `SET` 命令, 第一个命令将我们重定向到第一个节点, 这是因为通过计算 `crc16("book")` 值并对 16384 取模可以得到 `"book"` 哈希槽为 1337, 同时第二个键驻留在相同的节点上, 我们可以通过以下命令确认这一点:

```
127.0.0.1:9001> keys *
1) "{book}:3"
2) "book:1"
3) "{book}:4"
```

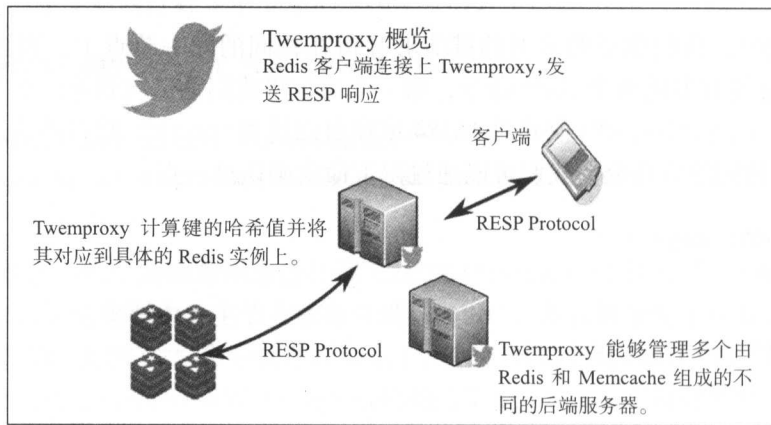
使用 Twemproxy 实现 Redis 集群

Twemproxy 是一个由 Twitter 发布的开源项目, 旨在创建客户端和由 Memcache 或者 Redis 实例组成的服务器端间的缓存代理。Twemproxy 通过采用中间件的方式, 分离客户端调用和数据存储后端, 在我们的示例中, 客户端指的是 Redis 客户端。该中间件基于 YAML 配置文件中配置的偏好设置实现了分区策略。Twemproxy 支持 12 种不同的哈希函数, 包括 `md5`、`crc16`、两种版本的 `crc32` 及 **FowlerNoll-Vo (FNV)** 的 4 种版本, 其中默认的哈希函数是 `fnv1a_64`。

Twemproxy 和 Redis 一样是由 C 语言编写的。要想运行 Twemproxy 需要一些不同的方法。为了让我们快点开始, 请到 <https://github.com/twitter/twemproxy/releases> 链接下载 Twemproxy 的发行 tar 包 (你同样可以下载源码 tar 包, 或者使用 Git 将仓库克隆下来。如果采用上述方式, 那么在运行以下 `configure` 命令前, 还需要额外运行 `autoconf` 命令才可以)。

```
$ tar xvf nutcracker-0.4.1.tar.gz
$ cd nutcracker-0.4.1/
$ ./configure
$ make
$ sudo make install
```

在运行 Twemproxy 之前，我们需要更新并配置代理以便使用 Redis，同时将运行中的 Redis 实例映射为 Twemproxy 的后端缓存服务器：



使用关联数据片段服务器测试 Twemproxy

为了开始使用关联数据片段服务器来测试 Twemproxy，后端的 Redis 集群将由 4 台 Redis 实例组成。2 台 Redis 实例组成主节点，负责关联数据缓存。剩余的 Redis 实例作为从节点，从主节点进行复制。我们为关联数据片段服务器项目添加并实现了一套轻量级 REST API。该项目是用专门用于搭建 REST API 的 Python 框架 Falcon 实现的，由 Rackspace 发布并维护。

在 Python 模块 `api.py` 里，用于实现 Triple REST 端点的一个新的类包含两个方法：其中 `on_get` 方法实现了 HTTP GET 方法调用，返回语法格式为 `{subject sha1}:{predicate sha1}:{object sha1}` 的键所对应三元组的简单 RDF 图，另一个 `on_post` 方法基于主语、谓语和宾语的 sha1 摘要来创建新的三元组，然后将值存储为整数 1。在客户端代码中，如果三元组键存在，那么首先将键分割成三个摘要值，分别对应主语、谓语和宾语，获取保存在 sha1 摘要键所对应的那些值，并在运行时构造用于返回的三元组的 JSON 关联数据表示。

为了比较关联数据片段服务器的性能,需要创建来源于以下两处的由基于 BIBFRAME 的 RDF 图所组成的测试数据:

- 所有匹配条件“Mark Twain”和“Bible”的国会图书馆 MARC 记录
- 科罗拉多学院图书馆中基于借阅次数统计得出所有最受欢迎资料的 MARC21 记录

这两个数据集一共代表了由超过 5,000,000 独立三元组所组成的超过 50,000 个不同的图。

我们将通过创建缓存目录,将 `cache.py` 移动到新的目录中并重命名为 `aioredis.py`,来扩展并继续隔离我们项目中基于 Redis 的代码。之后,我们将在相同的目录中创建新的 Python 模块并起名为 `twemproxy.py`。

为了开始测试,首先,我们需要修改 Twemproxy 中位于 `nutcracker-0.4.1/conf/nutcracker.yml` 的 YAML 配置文件。我们将创建一个简化的配置,为运行在从端口 6379 到 6383 的 Redis 节点使用单个服务器池 alpha。以下是 alpha 的 YAML 配置:

```
alpha:
  listen: 127.0.0.1:22121
  hash: fnv1a_64
  distribution: ketama
  auto_eject_hosts: true
  redis: true
  server_retry_timeout: 2000
  server_failure_limit: 1
  servers:
    - 127.0.0.1:6379:1
    - 127.0.0.1:6380:1
    - 127.0.0.1:6381:1
    - 127.0.0.1:6382:1
```

为了连接上 alpha,我们将使用 Twemproxy 端口 22121。在服务器配置中,我们可以看到列出的 Redis 实例映射到了其余的端口上。在 hash 选项中,我们选择了 `fnv1a_64`,它是 FNV 哈希函数的 64 位版本。FNV 哈希函数非常快速但是不适合用来加密,因为有可能暴力碰撞检测。Twemproxy 提供的其他哈希函数选择有 CRC (之前我们讨论过)等。对于哈希函数的选择依赖于计算速度和哈希碰撞的可能性。

alpha 中的 `distribution` 选项被设置为 `ketama`。这是一种哈希分布算法，将键哈希为圆上连续的无符号整数。每个数字关联到被用于哈希的服务器上。一个具体的键对应的整数被匹配到距离最近最大的数字上。当键对应的整数超过连续区间的最大值时就会回到圆上第一个数字。其他分布选项有 `modula`，它将键的值取模运算后对应到服务器上。还有 `random` 选项，它为键计算选择一台可用的 Redis 服务器。在启动四个 Redis 实例后，需要确保为每个 Redis 实例指定单独的端口和 RDB 文件名。我们将开启另一个命令行窗口并启动 Twemproxy：

```
$ ./src/nutcracker
[2015-08-17 06:30:52.957] nc.c:187 nutcracker-0.4.1 built for Darwin
14.0.0 x86_64 started on pid 626
[2015-08-17 06:30:52.958] nc.c:192 run, rabbit run / dig that hole,
forget the sun / and when at last the work is done / don't sit down /
it's time to dig another one
```

在 Twemproxy 运行起来之后，我们可以通过 `redis-cli` 连接上端口 22121 并发送命令：

```
$ redis/src/redis-cli -p 22121
```

为了在 Redis 实例中使用 Lua 脚本，我们将打开一个 Python 命令行，并遍历所有四个运行中的 Redis 实例，将 `add_get_triple.lua` 加载到每个实例中：

```
>>> import redis
>>> with open("/linked-data-fragments/redis/add_get_triple.lua") as fo:
add_get_triple = fo.read()
>>> cluster = []
>>> for port in range(6379, 6383):
    instance = redis.StrictRedis(port=port)
    instance.script_load(add_get_triple)
    cluster.append(instance)
```

让我们切回到连接在 Twemproxy 的 `redis-cli` 会话上，用 `add_get_hash` 的 `sha1` 哈希来调用 `EVALSHA`，结果如下所示：

```
127.0.0.1:22121> EVALSHA a5bb6a5952e578bdd2ddd9ede268ab28c6b90eb4 3
http://example.com/book/1 http://schema.org/name "Origins Reconsidered"
"2c866521408acafb64b0e67d17822260d68aadde:30cd0bd17373373839fb3a0ffaa6bba
51a17ba6c:574dbf58ad0e51382993cadec21742ae4de5aef8"
```

在 Twemproxy 对 Lua 脚本求值后, 返回的字符串是由 KEYS 变量中的每个值的 SHA1 组成的三元组。但是, 如果我们尝试获取 sha1 键 2c866521408acafb64b0e67d17822260d68aadde 所对应的值时, 我们会在 redis-cli 会话中收到 nil 值:

```
127.0.0.1:22121> GET 2c866521408acafb64b0e67d17822260d68aadde
(nil)
```

为什么是这样的结果? 这是因为 Lua 脚本为每个 RDF 三元组中的主语、谓语和宾语的键填写了值, 它绕过了 Twemproxy, 即使我们直接连接的是第一个 Redis 实例。我们可以确定这三个键只在一个实例中被设置了, 没有被 Twemproxy 代理, 如下所示:

```
127.0.0.1:6379> KEYS *
1) "30cd0bd17373373839fb3a0ffaa6bba51a17ba6c"
2) "2c866521408acafb64b0e67d17822260d68aadde"
3) "574dbf58ad0e51382993cadec21742ae4de5aef8"
127.0.0.1:6379> MGET 2c866521408acafb64b0e67d17822260d68aadde
30cd0bd17373373839fb3a0ffaa6bba51a17ba6c
574dbf58ad0e51382993cadec21742ae4de5aef8
1) "http://example.com/book/1"
2) "http://schema.org/name"
3) "Origins Reconsidered"
```

在关联数据片段服务器中使用 Twemproxy 意味着当前用于创建和填充 RDF 三元组的 Lua 脚本是不可能实现的。因此, Lua 脚本中, 的那些逻辑需要添加到 twemproxy.py 模块中。由于这段逻辑添加在了 Redis 缓存的最初实现中, 但是后来采用 Lua 脚本实现时将这段逻辑移除了, 所以我们将 sha1 哈希逻辑添加回 twemproxy.py 模块中。这里展示的是在项目中采用 Twemproxy 和 Redis 的关键之处, 也就是客户端代码和缓存服务器之间的交互必须通过代理, 客户端代码不能直接写入服务器实例:

关联数据片段服务器和 Twemproxy

Linked Data Fragments Server

输入的 URI 或者字面值的 SHA1 哈希摘要是要用作 RDF 三元组中主语、谓词和宾语这些原始值的键的。三元组键是由冒号分隔的哈希摘要并将值设置为 1



Subject: <http://catalog.coloradocollege.edu/435225>

Subject SHA1: `eff7da32770056f1e6a22657a016b2aaf93c9bdc`

Predicate: <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>

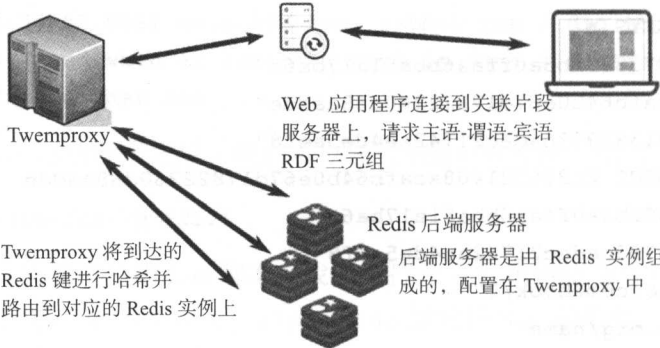
Predicate SHA1: `3c197cb1f6842dc41aa48dc8b9032284bcf39a27`

Object: <http://bibframe.org/vocab/Work>

Object SHA1: `5d1377f4476a1cbfb3cae106dc6b0a7d086410a`

Triple Key:

`eff7da32770056f1e6a22657a016b2aaf93c9bdc:3c197cb1f6842dc41aa48dc8b9032284bcf39a27:5d1377f4476a1cbfb3cae106dc6b0a7d086410a`



在往 `twemproxy.py` 中添加了额外的代码来为每个三元组创建 sha1 哈希之后, 我们来重新测试 Twemproxy 的性能。首先, 我们将创建新的 Lua 脚本 `get_triple`, 它接收一个三元组作为参数并返回以 JSON-LD 表示的三元组字符串:

```
local subject_shal, predicate_shal, object_shal = split(KEYS[1], ":")
local output = '{"@id": "'
output = output..redis.pcall('get', subject_shal)..'",'
output = output..redis.pcall('get', predicate_shal)..'":[{'
local object = redis.pcall('get', object_shal)
if string.sub(object,1,string.len("http")) == 'http' then
    output = output.."@id": "'
else
    output = output.."@value": "'
end
output = output..'"..object..""]}]}'
return output
```

接下来,我们将科罗拉多学院 MARC21 测试记录添加到运行着四台 Redis 实例的 Twemproxy 中。加载完成后,我们将使用 redis-clie 链接到每个 Redis 实例上,检测对应实例上的内存使用量:

```
127.0.0.1:6379> DBSIZE
(integer) 903287
127.0.0.1:6379> INFO memory
# Memory
used_memory:176939920
used_memory_human:168.74M
127.0.0.1:6380> DBSIZE
(integer) 836812
127.0.0.1:6380> INFO memory
# Memory
used_memory:164487104
used_memory_human:156.87M
127.0.0.1:6381> DBSIZE
(integer) 942231
127.0.0.1:6381> INFO memory
# Memory
used_memory:184067520
used_memory_human:175.54M
127.0.0.1:6382> DBSIZE
(integer) 879448
127.0.0.1:6382> INFO memory
# Memory
used_memory:172414320
used_memory_human:164.43M
```

我们的 Twemproxy 关联数据片段服务器上总共 3,561,778 个键,四个实例总共使用了 665.58 MB 内存。在 Redis 集群被开发和发布在 beta 测试和最终产品 (Redis 3) 之前, Twemproxy 是集群 Redis 数据的推荐方法。随着 Redis 集群的发布与测试的呼声越来越高,而 Twemproxy 却没有做什么动作,使用 Redis 集群来替代 Twemproxy 会更好。

Redis 集群的背景

Redis 集群始于 Salvatore Sanfilippo 在 2011 年在 Redis 邮件列表中的声明，以及后续一系列的博客文章。详情请参考书后附录第 6 章（可伸缩性：Redis 集群和 Sentinel 中第四个要点）。支持 Redis 集群的讨论起始于 2010 年，首次提到该术语是在一封发送给 Redis-db listserv 的电子邮件中。详情请参考附录第 6 章（可伸缩性：Redis 集群和 Sentinel 中第五个要点）。针对 Redis 集群的开发和测试从 2011 年一直持续到 2015 年。在 2014 年 10 月后续的博客文章中，Sanfilippo 叙述了自从他首次于 2011 年 3 月涉及 Redis 集群起，这 4 年多来他必须重新设计、实现，并且基于各种场景来测试 Redis 集群的功能。同时，他越发熟悉来自大规模分布式计算的挑战。在那一段的开发岁月里，Redis 社区不断尝试，但未能有效地解决两个问题。第一个问题是如何将数据分区到 N 个 Redis 实例中去。第二个问题是如何优雅地处理某些情况下的节点故障。为了应对第二个问题，Sanfilippo 开始着手 Redis 监控和故障转移工作，并最终造就了 Redis 的高可用性解决方案即 Redis Sentinel，也就是本章的最后一个主题。

虽然 Redis 集群提供了为数据集扩容的机制，并且可以由 Redis 进行管理，但是 Redis 集群并不提供健壮的一致性保证，也就是说在传播数据时数据可能会丢失。作为一个分布式系统，到达主节点的写操作，会在主节点传播数据到从节点之前将确认消息立即发送给调用的客户端。如果在将新的数据传播到从节点之前主节点发生故障，而同时从节点提升为主节点的话，那么该数据会丢失。

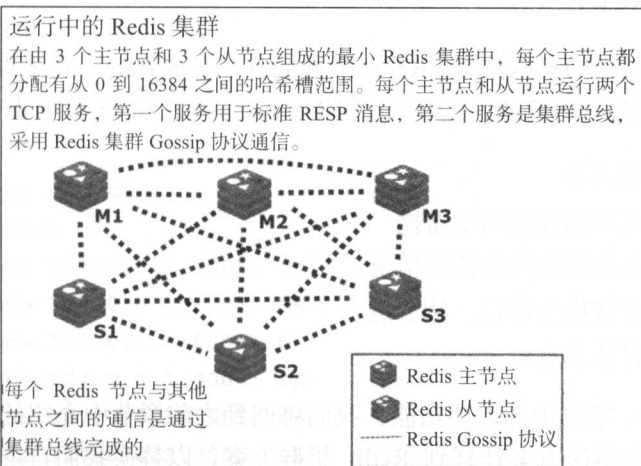
Redis 集群默认运行的传播模式是异步的流程（在与客户端交互期间，Redis 集群会继续写操作），在主从节点的配置下，集群的性能胜于数据的一致性。如果数据一致性对于应用程序来说比性能更重要，那么 Redis 集群提供的 WAIT 命令可以用来将数据传播的方式变更成同步流程。在此设定下，客户端会一直阻塞直到特定数量的从节点确认写操作，或者发生了超时为止。使用 WAIT 命令只是增进了 Redis 集群数据一致性和安全性，但是仍然无法保证作为分布式数据存储环境下的数据强一致性。

在 Redis 集群中设置配置 `node timeout` 指令意味着如果主节点未能在指定的时间窗口内进行响应，那么其他主节点就会认为该主节点发生了故障，同时该主节点会被对应的从节点替换。在该场景中，原始的主节点停止接受写请求，因为它没有接收来自集群中其他主节点的通信。

Redis 集群概览

在 Redis 集群中的 Redis 实例要么是主节点, 要么是从节点。每一个 Redis 主节点被分配了 16384 个哈希槽中的一到多个。Redis 键根据其 CRC16 的计算结果再对主节点数取余最终分配到一个哈希槽上。当 Redis 集群在运行时, 当中的每个节点会打开两个 TCP socket。第一个用于连接客户端的标准 Redis 协议, 默认端口是 6379。第二个端口是由第一个端口加上 10000 (对于默认端口来说就是 16379 了) 计算所得, 运行着用于节点间通信的集群二进制协议。客户端从来不必直接连接到集群的总线端口上, 而是应该连接到更低的标准端口上。Redis 集群中的节点使用 Redis 集群总线在网状网络拓扑结构中与其他节点连接。这也意味着对于由 3 个主节点和 3 个从节点共 6 个节点组成的 Redis 集群来说, 每个节点除了它自己的复制状态外, 拥有 5 条对外 TCP 连接及 5 条对内 TCP 连接。这些连接一直是活跃的, 并且持续响应来自集群中其他节点的 ping 消息。这些称作心跳包 (Heartbeat Packets) 的消息, 包括 Node ID, currentEpoch, 节点标志位, 发送方所服务的哈希槽位图, TCP 基本端口, 发送方拥有的集群状态视角 (正常运行, 故障中, 已故障), 以及主节点 ID (如果发送方为从节点时)。

为了避免集群中节点间消息呈指数型增长, 集群采用了 Gossip 协议, 以及包含了集群网络拓扑中的所有节点间发送的消息总数的配置更新过程。



虽然集群中的任意一个节点都会接受来自 Redis 集群总线发来的 TCP 连接请求, 但只有当请求的节点是集群中的组成部分时, 节点才会回应消息而不仅仅是确认响应。有两种方法能够让节点被认为是 Redis 集群中的组成部分。第一种方法是, 如果第一个节点发送 MEET 消息给第二个节点, 那么第二个节点必须将第一个节点视为集群的组成部分。MEET

命令是通过发送 `CLUSTER MEET` 命令进行设置的。第二种方法是基于节点间逻辑传递关系的 gossip 算法。如果节点 1 和节点 2 均是集群的组成部分，并且节点 2 知道节点 3，那么最终节点 1 会与节点 2 交换有关节点 3 的 gossip 消息，于是节点 1 将节点 3 注册为 Redis 集群的组成部分。这样的方案允许动态自动发现集群中的其他节点，同时又提供了更健壮的 Redis 集群，避免在集群运行时协调添加新节点所带来的巨大开销。

Redis 集群上下文中 *epoch* 解释了 Redis 集群是如何在整个 Redis 集群运行生命周期内创建版本历史的。Epoch 是一个 64 位无符号整数，当发生像添加新的主节点这样的节点事件时该值会增加。*epoch* 存储在 `currentEpoch` 变量里。在集群初始化时，所有的主节点和从节点都设置为 0。当使用 Redis 的 gossip 协议收到一条头部带有 *epoch* 的消息时，如果接收该消息的节点的 *epoch* 值小于消息中的 *epoch* 值，那么该 Redis 节点将 `currentEpoch` 更新为最高的 *epoch* 值。在这种方式下，集群最终认同 *epoch* 的最高值，并提供显著改变集群中主节点基本组成的事件的线性路径。当从节点因 *epoch* 中的故障转移事件而提升为主节点时，意味着从节点能被增量地添加到不同的运行中的虚拟机上而不会导致 *epoch* 值更新。但当从节点提升为新的主节点时，因为这种提升会潜在地改变整个节点的运行时特征或者在之后导致未知的结果，所以显得十分重要。

使用 Redis 集群

作为企业产品级别的集群方案，Redis 集群拥有一套成熟的工具来管理运行时 Redis 集群节点。Redis 集群包含的 Ruby 实用工具脚本 `redis-trib.rb`，主要支持下列功能：

- 重新分片与故障转移
- 在节点中移动或者创建新的哈希槽
- 处理诸如主节点故障这样的错误情况
- 集群中添加或者替换主节点、从节点
- 升级主节点或者从节点

为了测试 Redis 集群中的上述功能，我们将回到之前章节中介绍的区域代码示例中，并将我们基于列表的解决方案迁移到 Redis 集群方案，以替换我们自定义的客户端代码。依据 <http://redis.io/topics/cluster-tutorial> 上官方 Redis 教程的示例，我们将运行推荐的最小 Redis 集群，包含了 3 个主节点和 3 个从节点。包含在 Redis 内的实用工具 `create-cluster` 是创建并运行 Redis 集群最简单的方式，我们将在区域代码示例中使用该工具。首先，我们创建新的 `config.sh` 文件指定具体的 Redis 集群选项：

```
$ cd redis/utils/create-cluster
$ vi config.sh
#!/bin/bash
PORT=9000
TIMEOUT=2000
NODES=6
REPLICAS=1
```

我们的集群配置了 9000 端口, 2 秒钟的延迟, 由 3 个主节点和 3 个从节点组成的 6 个节点。如果我们想要增加副本数量, 那么增加 replicas 选项就意味着每个主节点将拥有 replicas 数量的从节点。

在保存 config.sh 之后, 我们需要先来安装 Redis gem:

```
$ sudo gem install redis
```

现在, 我们运行 create-cluster 脚本来创建并启动集群:

```
$ cd ~/redis/util/create-cluster
$ ./create-cluster start
Starting 9001
Starting 9002
Starting 9003
Starting 9004
Starting 9005
Starting 9006
$ ./create-cluster create
>>> Creating cluster
Connecting to node 127.0.0.1:9001: OK
Connecting to node 127.0.0.1:9002: OK
Connecting to node 127.0.0.1:9003: OK
Connecting to node 127.0.0.1:9004: OK
Connecting to node 127.0.0.1:9005: OK
Connecting to node 127.0.0.1:9006: OK
>>> Performing hash slots allocation on 6 nodes...
Using 3 masters:
127.0.0.1:9001
127.0.0.1:9002
127.0.0.1:9003
```

```

Adding replica 127.0.0.1:9004 to 127.0.0.1:9001
Adding replica 127.0.0.1:9005 to 127.0.0.1:9002
Adding replica 127.0.0.1:9006 to 127.0.0.1:9003
M: 9ed33dd148ba6546431b2439d1e85b3b742ef336 127.0.0.1:9001
  slots:0-5460 (5461 slots) master
M: de3ec68f65de532080e296be3a2b1502e35fe281 127.0.0.1:9002
  slots:5461-10922 (5462 slots) master
M: c12d7eae35befeb8530d6fec366fb34aaed9eefc 127.0.0.1:9003
  slots:10923-16383 (5461 slots) master
S: 623b9338e6fc277634a741e7f56c8a08240ff7d0 127.0.0.1:9004
  replicates 9ed33dd148ba6546431b2439d1e85b3b742ef336
S: f6efd99a0505072ca3539a629674eb88ffaaa78f 127.0.0.1:9005
  replicates de3ec68f65de532080e296be3a2b1502e35fe281
S: b9656e82c01ca7d085b6386d7ca8383903897157 127.0.0.1:9006
  replicates c12d7eae35befeb8530d6fec366fb34aaed9eefc
Can I set the above configuration? (type 'yes' to accept): yes

```

运行 `create-cluster` 命令首先会连接所有 6 个运行中的节点, 然后在 3 个主节点上进行哈希槽分配。这三个主节点分别运行在端口 9001、9002 和 9003 上。在配置完之后, `create-cluster` 脚本所使用的 `redis-trib.rb` 脚本会输出以下内容:

```

>>> Nodes configuration updated
>>> Assign a different config epoch to each node
>>> Sending CLUSTER MEET messages to join the cluster
Waiting for the cluster to join.
>>> Performing Cluster Check (using node 127.0.0.1:9001)
M: 9ed33dd148ba6546431b2439d1e85b3b742ef336 127.0.0.1:9001
  slots:0-5460 (5461 slots) master
M: de3ec68f65de532080e296be3a2b1502e35fe281 127.0.0.1:9002
  slots:5461-10922 (5462 slots) master
M: c12d7eae35befeb8530d6fec366fb34aaed9eefc 127.0.0.1:9003
  slots:10923-16383 (5461 slots) master
M: 623b9338e6fc277634a741e7f56c8a08240ff7d0 127.0.0.1:9004
  slots: (0 slots) master
  replicates 9ed33dd148ba6546431b2439d1e85b3b742ef336
M: f6efd99a0505072ca3539a629674eb88ffaaa78f 127.0.0.1:9005
  slots: (0 slots) master
  replicates de3ec68f65de532080e296be3a2b1502e35fe281

```

```
M: b9656e82c01ca7d085b6386d7ca8383903897157 127.0.0.1:9006
```

```
slots: (0 slots) master
```

```
replicates c12d7eae35befeb8530d6fec366fb34aaed9eefc
```

```
[OK] All nodes agree about slots configuration.
```

```
>>> Check for open slots...
```

```
>>> Check slots coverage...
```

```
[OK] All 16384 slots covered.
```

为了验证我们确实运行了包含 3 个主节点和 3 个从节点共 6 个节点, 我们将启动 redis-cli 会话, 并携带特殊的参数 -c, 以便在不同的主节点哈希槽进行切换:

```
$ ../../src/redis-cli -c -p 9001
```

```
127.0.0.1:9001>
```

发送 CLUSTER INFO 命令就能展示运行中的集群状况:

```
127.0.0.1:9001> CLUSTER INFO
```

```
cluster_state:ok
```

```
cluster_slots_assigned:16384
```

```
cluster_slots_ok:16384
```

```
cluster_slots_pfail:0
```

```
cluster_slots_fail:0
```

```
cluster_known_nodes:6
```

```
cluster_size:3
```

```
cluster_current_epoch:6
```

```
cluster_my_epoch:1
```

```
cluster_stats_messages_sent:5430
```

```
cluster_stats_messages_received:5430
```

发送 CLUSTER NODES 命令可以获取运行中节点的具体信息:

```
127.0.0.1:9001> CLUSTER NODES
```

```
b9656e82c01ca7d085b6386d7ca8383903897157 127.0.0.1:9006 slave
```

```
c12d7eae35befeb8530d6fec366fb34aaed9eefc 0 1439299522348 6 connected
```

```
f6efd99a0505072ca3539a629674eb88ffaaa78f 127.0.0.1:9005 slave
```

```
de3ec68f65de532080e296be3a2b1502e35fe281 0 1439299522348 5 connected
```

```
de3ec68f65de532080e296be3a2b1502e35fe281 127.0.0.1:9002 master - 0
```

```
1439299522348 2 connected 5461-10922
```

```
c12d7eae35befeb8530d6fec366fb34aaed9eefc 127.0.0.1:9003 master - 0
```

```
1439299522348 3 connected 10923-16383
9ed33dd148ba6546431b2439d1e85b3b742ef336 127.0.0.1:9001 myself,master - 0
0 1 connected 0-5460
623b9338e6fc277634a741e7f56c8a08240ff7d0 127.0.0.1:9004 slave
9ed33dd148ba6546431b2439d1e85b3b742ef336 0 1439299522348 4 connected
```

现在，我们将创建一个随机电话号码生成器来填充我们的集群。现实中，这些数据更有可能来自于 CRM 应用或者其他客户数据来源。我们将使用现有的北美区域代码数据创建十进制随机电话号码生成器：

```
>>> import random
>>> def random_phonenumber(area_code):
    number = str(area_code)
    for i in range(7):
        number += "{}".format(random.randint(0,9))
    return number
```

我们将使用并导入 Redis 集群 Python 库 `redis-py-cluster`（可以从 <https://github.com/Grokzen/redis-py-cluster> 上进行下载）。之后，我们实例化 Python Redis 集群来填充区域代码应用：

```
>>> from rediscluster import StrictRedisCluster
>>> startup_nodes = [{"host": "localhost", "port": 9001}]
>>> area_code_cluster = StrictRedisCluster(startup_nodes=startup_nodes)
```

然后我们将创建第二个函数，用它来导入名为 `area-code.txt` 的 CSV 文件，并返回由区域代码映射到地理区域的字典。首先迭代文本文件对象的每一行，创建一个字段列表，然后将区域代码转换为整型，并用字段列表的第二个元素给 Python 字典赋值：

```
>>> def area_code_dict(filepath):
    with open(filepath) as fo:
        lines = fo.readlines()
        area_codes = dict()
        for row in lines:
            fields = row.split("\t")
            area_codes[int(fields[0])] = fields[1].strip()
    return area_codes
```

最后一个函数从区域代码 Python 字典的代码键列表中随机选取一个区域代码,调用之前定义好的 `random_phonenumber` 函数,然后将该(随机生成的)电话号码作为 Redis 哈希值保存到运行中的 Redis 集群里,同时将 `geographicArea` 域设置为之前随机到的 `area_codes` 字典的那个值:

```
>>> def populate_cluster(total):
    codes = list(area_codes.keys())
    for i in range(total):
        number = random.randint(0, len(codes)-1)
        area_code = codes[number]
        phone_number = random_phonenumber(area_code)
        area_code_cluster.hsetnx(phone_number, "geographicArea", area_
codes[area_code])
>>> populate_cluster(150000)
```

之后,我们用 150000 随机电话号码填充 Redis 集群。本章的测试运行结果分析如下(如果你自行重复这些练习,你的哈希槽中键的分布会有所不同):

	主节点	从节点	大 小
第一个哈希槽	127.0.0.1:9001	127.0.0.1:9004	50005
第二个哈希槽	127.0.0.1:9002	127.0.0.1:9005	49971
第三个哈希槽	127.0.0.1:9003	127.0.0.1:9006	50020

从这些结果中我们可以看到,那些电话号码几乎均匀地分布到了所有 3 个哈希槽中,三个主节点的总方差为 49,差异不到 1%。使用这些区域代码作为我们的测试集群,我们现在进行一系列的联系人来展示 Redis 集群中一些重要的操作和功能。

Redis 集群实时重新配置及重新分片

Redis 集群提供了不少命令用来在运行时添加和删除节点。当向集群中添加新的空节点时,首先需要将该节点添加到集群里,然后将现有节点中至少一个槽重新分配给该新节点。

`CLUSTER ADDSLOTS` 命令主要用来手工创建 Redis 集群并分配 16384 的可用哈希槽,而 `CLUSTER DELSLOTS` 命令则用来手工修改集群或者用于测试和调试。

`CLUSTER SETSLOT` 命令在不同的场景下有多种用法。像 `CLUSTER SETSLOT {哈希槽编号} NODE {节点 id}` 这样的形式是用来在受限的条件下将槽分配给具体的节点。此外, `CLUSTER SETSLOT` 接受 `MIGRATING` 或者 `IMPORTING` 形式的参数,这取决于

你是想要 MIGRATING 目的节点还是 IMPORTING 源节点。对像 `CLUSTER SETSLOT {哈希槽编号} MIGRATING {目的节点 id}` 这样的命令和子命令来说，节点将继续接受符合被分配哈希槽的查询请求，但对于那些不符合要求的请求会被重定向到另一个节点的新哈希槽。如果命令包含多个键，那么结果将依赖于键是否存在，或者发出一个 TRYAGAIN 错误。

同样，当 `CLUSTER SETSLOT` 被设置为 IMPORTING 模式时，源节点会拒绝该哈希槽的任何请求，它在收到 ASKING 命令之前会发送 MOVED 重定向。ASKING 命令为客户端设置了一次性标志，并强制客户端仅将下一次请求发送到具体的节点，而不是像 MOVED 错误那样永久性地将客户端重定向到被分配了哈希槽的那个节点上。

为了测试这些实时配置命令，我们将在 9007 端口启动新的 Redis 主节点，并将其配置指令 `cluster-enabled` 设置为 yes，然后我们将添加新的节点到我们的区域代码集群中，只需在新的 `redis-cli` 会话中使用 `CLUSTER MEET` 命令：

```
127.0.0.1:9007> CLUSTER MEET 127.0.0.1 9002
OK
```

我们可以通过再次运行 `CLUSTER INFO` 命令确认新的节点成为了集群的组成部分，同时注意到 `cluster_known_nodes` 的值：

```
127.0.0.1:9007> CLUSTER INFO
cluster_state:ok
cluster_slots_assigned:16384
cluster_slots_ok:16384
cluster_slots_pfail:0
cluster_slots_fail:0
cluster_known_nodes:7
cluster_size:3
cluster_current_epoch:6
cluster_my_epoch:3
cluster_stats_messages_sent:1037
cluster_stats_messages_received:1037
```

我们将手动转移一个槽，为将键迁移到新的主节点上做准备。该主节点有一个随机 ID `ed862747677b458cf6c79f58b29b4e4c09a9603b`：

```
127.0.0.1:9007> CLUSTER SETSLOT 12000 NODE
ed862747677b458cf6c79f58b29b4e4c09a9603b
OK
```

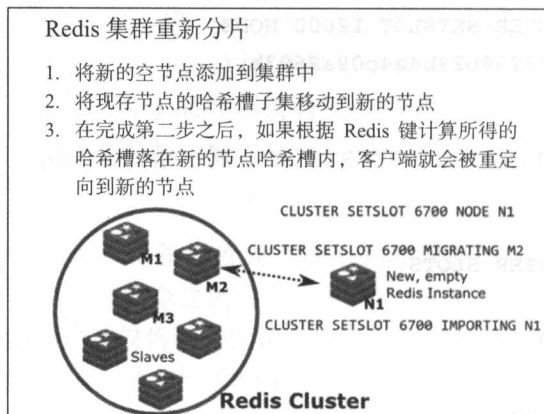
为了仔细检查,我们运行 `CLUSTER SLOTS` 展示槽的分配情况(虽然没有键被迁移到新的主节点上):

```
127.0.0.1:9007> CLUSTER SLOTS
```

```
1) 1) (integer) 0
   2) (integer) 5460
   3) 1) "127.0.0.1"
      2) (integer) 9001
   4) 1) "127.0.0.1"
      2) (integer) 9004
   2) 1) (integer) 10923
2) (integer) 16383
   3) 1) "127.0.0.1"
      2) (integer) 9003
   4) 1) "127.0.0.1"
      2) (integer) 9007
   5) 1) "127.0.0.1"
      2) (integer) 9006
   3) 1) (integer) 5461
      2) (integer) 10922
3) 1) "127.0.0.1"
   2) (integer) 9002
   4) 1) "127.0.0.1"
   2) (integer) 9005
```

从以上输出中我们可以看到,新的节点确实被分配到了正确的哈希槽范围。

虽然当前 Redis 集群中基于 CRC16 的组合哈希能够差强人意地将键平均地分发到所有的主节点上,但是实际应用的数据还是需要你亲自将集群重新分片,以便更好地平衡集群节点间的负载。幸运的是,使用 Redis 附带的基于 Ruby 的 `redis-trib.rb` 实用工具可以简化这一过程:



故障转移

Redis 集群运行着多个主节点并且每个主节点对应一到多个从节点。正常情况下, 当一个主节点故障时, 它的任何一个从节点会被自动选举出来替换主节点。当从节点所属主节点处于 FAIL 状态, 并且该主节点被分配有一个或多个哈希槽, 同时主从节点间最后一次成功连接的时间长度低于使用 `cluster-slave-validity-factor` 配置指令计算所得的临界值时, 会启动从节点选举。当一个主节点拥有多个从节点时, 这些从节点之间会计算出等级。拥有最完整的主节点的内容复制的从节点将被赋予最高等级。在从节点提升为新的主节点之后, 集群的 *epoch* 会递增, 同时新的主节点的变更将通过 *gossip* 协议传播到集群中其他的节点。

Redis 集群提升运行时集群可靠性的一个重要方面是 Redis 主节点的从节点分布情况。在我们这个简单的三个主节点和三个从节点的设置中, 由于主节点故障而促使从节点替换主节点, 这意味着新的主节点没有备用从节点实例。如果新的主节点故障, 集群将无法从故障主节点对应的哈希槽读取或者写入数据, 因而集群本身会发生故障, 同时那些哈希槽内的数据将无法再被访问, 或者更糟的是数据永久丢失。在更持久化的配置中, 主节点拥有多个从节点, 这时如果主节点故障, 级别最高的从节点就提升为新的主节点, 剩余的其他从节点则切换并成为新提升主节点的从节点。

为了展示 Redis 集群故障转移模式, 我们将回到区域代码示例中, 不过这次我们会为每个主节点添加 2 个从节点, 将 `config.sh` 文件中的集群节点数设置为 9。我们运行以下 `create-cluster` 脚本来启动一个空的 Redis 集群:

```
$ ./create-cluster clean
```

现在,我们将发送 `create-cluster start` 和 `create-cluster create` 命令启动新的由 9 个节点组成的 Redis 集群。我们会再次执行 `populate_cluster` 函数,用 15000 个北美电话号码来填充集群,这些号码被分区至三个主节点实例上,只不过这些主节点背后有了 2 个从节点。

在集群填充完毕后,我们使用 `-c` 参数启动 `redis-cli` 会话连接至集群节点,发送 `CLUSTER SLOTS` 命令展示运行中的主节点和两个从节点:

```
127.0.0.1:9001> CLUSTER SLOTS
```

```
1) 1) (integer) 5461
    2) (integer) 10922
    3) 1) "127.0.0.1"
        2) (integer) 9002
    4) 1) "127.0.0.1"
        2) (integer) 9007
    5) 1) "127.0.0.1"
        2) (integer) 9006
2) 1) (integer) 0
    2) (integer) 5460
    3) 1) "127.0.0.1"
        2) (integer) 9001
    4) 1) "127.0.0.1"
        2) (integer) 9004
    5) 1) "127.0.0.1"
        2) (integer) 9005
3) 1) (integer) 10923
    2) (integer) 16383
    3) 1) "127.0.0.1"
        2) (integer) 9003
    4) 1) "127.0.0.1"
        2) (integer) 9008
    5) 1) "127.0.0.1"
        2) (integer) 9009
```

现在,我们将打开第二个 `redis-cli` 会话连接至运行在 `127.0.0.1:9003` 上的主节点,该主节点分配到了 10823 到 16383 的哈希槽。我们发送一个 `DEBUG SIGFAULT` 命令来模拟主节点故障:

```
127.0.0.1:9003> DEBUG SEGFAULT
Could not connect to Redis at 127.0.0.1:9003: Connection refused
(0.90s)
not connected>
```

切换回原来的那个 redis-cli 会话，如果我们重新发送 CLUSTER SLOT 命令，我们将看到运行在端口 9003 上的原始主节点不出现了，同时运行在端口 9009 上的从节点被选举为新的主节点（为了清晰，我们忽略其他从节点）：

```
127.0.0.1:9001> CLUSTER SLOTS.
```

```
.
3) 1) (integer) 10923
    2) (integer) 16383
    3) 1) "127.0.0.1"
    2) (integer) 9009
    4) 1) "127.0.0.1"
    2) (integer) 9008
```

为了提升集群的灵活性，Redis 集群实现了称为副本迁移的算法，它会在主节点没有任何从节点的情况下，将从节点重新分配给该主节点。回顾我们之前的区域代码示例，如果我们再次遭遇服务于 10923~16383 哈希槽的主节点故障，那么运行在 9008 端口上的最后一个从节点将提升为主节点。现在，这个新的主节点没有从节点，因此 Redis 集群会将其他主节点的一个从节点迁移给这个新的主节点，以便集群中的每个节点都能至少拥有一个从节点。这样，Redis 集群通过副本迁移策略能够最终确保集群中的每个主节点至少拥有一个从节点。通常情况下，拥有多个从节点的主节点只会将一个从节点迁移给缺少从节点的主节点。不过，可以通过调整 cluster-migration-barrier 配置指令更改这一行为，该指令限制了集群中从节点能够迁移到 Redis 主节点的数量。

在 Redis 集群中替换或者升级节点

也许有一天，你会需要手动替换 Redis 集群中的主节点或者从节点。Redis 集群自动化的故障转移过程，也就是我们在上个章节中看到的那样将从节点提升为主节点，它还不足以满足应用程序的运营需求。

在长期运行的应用程序中，Redis 在企业级中表现出色。也许有一天，关键 BUG 的修复或者只是想让运行中的 Redis 集群使用最新稳定版本的 Redis，这些原因会促使你更新 Redis 的版本。升级集群中 Redis 节点的过程与替换节点的过程相似。首先，将 CLUSTER

FAILOVER 命令发送给目标主节点的一个从节点,然后将老的主节点转变为从节点,同时将从节点提升为主节点。其次,老的主节点(即现在的从节点)被新升级的节点替换。

当在 Redis 集群中替换或升级节点时,也可以发送 CLUSTER RESET 命令,并带上 SOFT 或者 HARD 参数,以便移除旧的主节点或者重新分配其哈希槽。如果目标节点是从节点,那么该命令会将这个节点转变为主节点而忽略在此过程中的任何数据。CLUSTER RESET 命令会清除所有分配给该节点的哈希槽,同时清除的还有节点表中的其他节点,因而该节点将对集群中其他节点的状态一无所知。在 HARD 模式下,节点的 epoch 变量(currentEpoch、configEpoch 及 lastVoteEpoch)都会被重置为 0,同时会分配一个新的随机节点 ID。

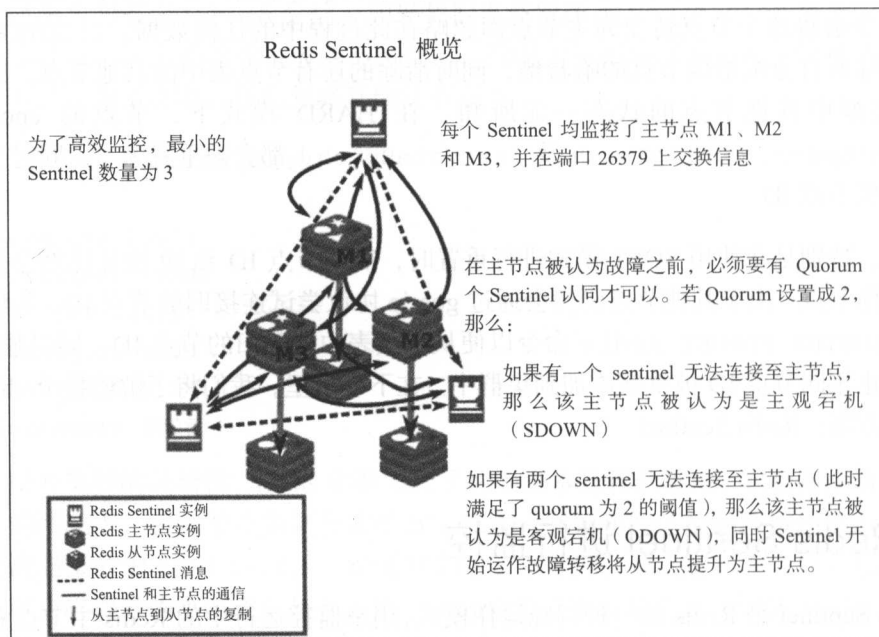
最后,特别是在使用 SOFT 模式进行重置时,旧的节点 ID 和 IP 地址仍然会被存储于集群中其他节点。由于其他节点仍然会通过 gossip 协议尝试连接旧的节点 ID,我们可以发送一个 CLUSTER FORGET Redis 命令以便从节点表中移除旧的节点 ID,同时强制 60 秒超时以阻止相同节点 ID 被重新添加到集群中。在下一节里,我们将了解监控 Redis 时更通用的解决方案: Redis Sentinel。

使用 Redis Sentinel 进行监控

Redis Sentinel 是 Redis 的一种特殊运作模式,用来监控运行中的 Redis 主节点和从节点实例。Redis Sentinel 会将故障的主节点实例替换为作为副本的从节点。同时其他类型的故障转移选项给了 Redis 用户更多高可用性的选择。除了监控和自动故障转移之外,Redis Sentinel 也提供了通知选项。当所监控的 Redis 实例发生严重错误时,Redis Sentinel 会通过 API 向系统管理员或者其他程序发送告警。同时,Redis Sentinel 能够为客户端自动发现服务提供配置以协助运维管理。理论上很简单,实际上 Redis Sentinel 会被应用在企业级的极其复杂的工作流程,以及监控/故障转移的使用场景中。

作为一个分布式系统,Redis Sentinel 需要至少部署三个实例以满足健壮要求。多个 Sentinel 间通过协作来检测发生故障的主节点,依据的是 Sentinel 中的大多数认为发生故障时才算发生故障。这在技术上可以减少关于主节点可用性的误报,同时也意味着即便主节点状态是健康的,仍然会将运作中的该节点剔除。如果其中有的 Sentinel 实例自身发生故障,多个 Redis Sentinel 间同样会以协作的方式保持正常运作,以避免监控系统发生故障。不同于 Redis 服务器,正常运行 Redis Sentinel 实例需要一份 sentinel.conf 配置文件。该文件包含在 Redis 的发行版中。

Sentinel 用于和其他 Redis Sentinel 通信的默认 TCP 端口是 26379。在运行 Redis Sentinel 实例的服务器上，该端口必须开启。理想情况下，每一个 Redis Sentinel 应用运行在独立的物理机或者虚拟机上以实现诸如可用性区域等不同的运维需求。这样做的目的是为了降低软件栈、物理硬件或者网络连接等原因导致单点故障的可能性。



使用 Redis Sentinel 并不保证数据上的强一致性，这是因为在绝大多数主/从和 Redis 集群设置中，Redis 本身使用了异步复制的方式。对 Redis Sentinel 合理的部署可以通过减少写操作丢失的时间窗口最小化写操作被丢弃的概率。应用程序中的 Redis 客户端应当清楚知道 Redis Sentinel，而且大多数流行的 Redis 客户端已经提供对 Redis Sentinel 的支持。

之前曾提到过，Redis Sentinel 的运行需要一份配置文件。在 `sentinel.conf` 范例文件中，有许多重要的配置指令用于自定义 Redis Sentinel 配置，这取决于 Redis 应用程序的组织与构成。首先，`sentinel monitor` 指令明确了该 Sentinel 实例将要监控哪个 Redis 主节点，同时需要设定 4 个参数：主节点的名称、主节点 IP 地址、端口号及 quorum 级别。

Redis Sentinel 自动监控 Redis 主节点的所有从节点。`down-after-milliseconds` 指令和子命令明确了在主节点被确认宕机前，Sentinel 实例无法连接到该主节点的毫秒数。Redis Sentinel 监控的每个主节点都需要进行设定。

Redis 集群 quorum



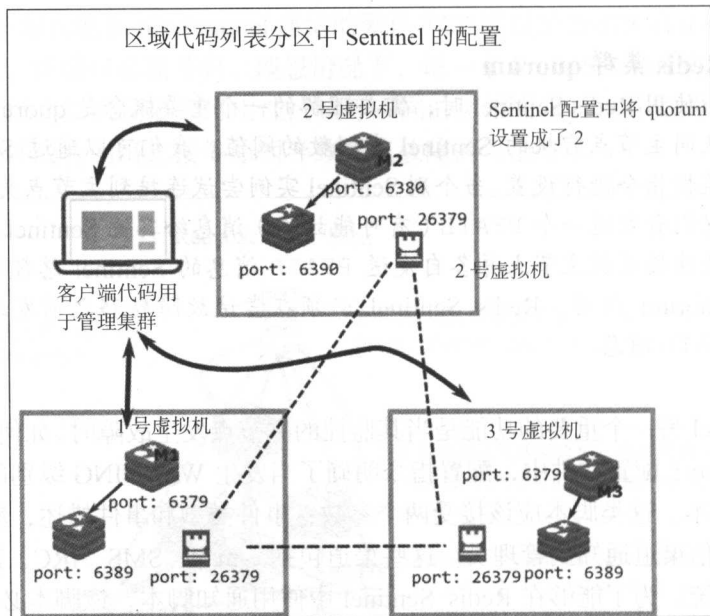
当使用 Redis Sentinel 时, 需要理解的一个重要概念是 quorum, 即认同主节点宕机的 Sentinel 实例数的阈值。我们可以通过 Sentinel 监控指令进行设置。当个别 Sentinel 实例尝试连接到主节点失败时, 它们会发送一个 PFAIL (有可能故障) 消息给其他 Sentinel。当无法连接至该主节点并各自发送 PFAIL 消息的 Sentinel 总数超过了 quorum 值时, Redis Sentinel 必须在尝试故障转移之前发送一条 FAIL 消息。

Redis Sentinel 另一个重要的功能是当其监控的主节点发生故障时, 如何设置消息通知。在 `sentinel.conf` 配置文件中, 配置指令明确了当发生 WARNING 级别的事件时运行的 `bash` 或者其他脚本。这类脚本应该接受两个参数: 事件类型和事件描述, 然后就能通过运维员工首选的通信渠道通知到管理员。这些渠道包括 e-mail、SMS、IRC, 甚至是通过 API 调用其他监控系统。为了能够在 Redis Sentinel 中使用通知脚本, 该脚本必须存在于 Redis Sentinel 能够访问的位置并且它的执行标志位必须设置成 `True`。

虽然配置并运行 Redis Sentinel 相对简单, 但是按照特定需求, 诸如性能权衡、理想化的持久性、网络分区和机器资源可用性等, 来决定 Redis Sentinel 的配置更富有挑战。Redis Sentinel 的最小化配置应当包括至少 3 个运行实例, 理想情况下会运行在单独的物理机器上, 包括那些专门用于为客户端应用程序服务的 Web、数据库和应用服务器的虚拟机。回顾本章中用到的几个示例, 现在我们将要概述如何为区域代码列表分区配置 Redis Sentinel。

为区域代码列表分区配置 Redis Sentinel

为区域代码列表分区示例配置的 Redis 节点由 3 个主节点组成。我们将为这些主节点添加 3 个从节点用于复制。为了简化部署, 每个主节点和其对应的从节点将运行在单独的虚拟机上。在此示例中, 我们的 Redis Sentinel 设置实现了最小化 Redis Sentinel 数目, 3 个 Sentinel 分别和主节点运行在同一台 VM 上, 并将 Redis Sentinel quorum 设置成 2:



在手工加载并启动每台测试虚拟机上的两个 Redis 实例及 Redis Sentinel 实例之后，我们将开启 redis-cli 会话并连接到 VM2 上的 Redis Sentinel 端口 26379（我们会截断部分结果，只展示一些重要的属性）：

```
127.0.0.1:26380> SENTINEL masters
```

```
1) 1) "name"
   2) "vm3"
   3) "ip"
   4) "172.26.6.145"
   5) "port"
   6) "6379"
   ...
  13) "last-ping-sent"
  14) "421859"
  15) "last-ok-ping-reply"
  16) "421859"
  ...
  25) "role-reported"
  26) "master"
  ...
```

```
29) "config-epoch"
30) "0"
31) "num-slaves"
32) "1"
...
33) "num-other-sentinels"
34) "3"
35) "quorum"
36) "2"
2) 1) "name"
   2) "vm2"
   3) "ip"
   4) "127.0.0.1"
   5) "port"
   6) "6380"
   ...
   13) "last-ping-sent"
   14) "0"
   15) "last-ok-ping-reply"
   16) "400"
   ...
   23) "role-reported"
   24) "master"
   ...
   27) "config-epoch"
   28) "0"
   29) "num-slaves"
   30) "1"
   31) "num-other-sentinels"
   32) "3"
   33) "quorum"
   34) "2"
   ...
3) 1) "name"
   2) "vm1"
   3) "ip"
   4) "172.29.40.33"
```

```
5) "port"
6) "6379"
...
13) "last-ping-sent"
14) "421859"
15) "last-ok-ping-reply"
16) "421859"
25) "role-reported"
26) "master"
...
29) "config-epoch"
30) "0"
31) "num-slaves"
32) "0"
33) "num-other-sentinels"
34) "3"
35) "quorum"
36) "2"
...
```

总结

Redis 3.x 系列版本中最大的改变在于包含了可工作的、稳定的、准产品特性的 Redis 集群。不管是对可伸缩性还是将数据分布到不同机器上的不同的 Redis 实例中，Redis 集群都是备受推崇的方法。Redis 集群实现了将传入键进行哈希计算的一种方法，即通过组合分区的方法，将哈希分区和范围分区的特性进行组合。同时，也有其他方法通过客户端分区的方式扩缩数据，其中一些在本章中介绍了。我们探索了一个流行的（开源的）用于分片和数据分区的产品。它就是 Twitter 的 Twemproxy。它提供了中间代理，负责处理应用和后台 Redis 服务实例间的哈希和分配逻辑。然后又回顾并仔细讲解了 Redis 集群的部分特性和功能，包括重新分片、故障转移、替换和升级选项，这些都有助于长期运行的 Redis 集群处理大容量的数据。最后，本章介绍了使用 Redis Sentinel 的部分高级用法，

在第 7 章中，我们将看到 Redis 能够与其他 NoSQL 技术形成很好的互补，为应用程序提供完整的解决方案。

7

Redis 与互补的 NoSQL 技术

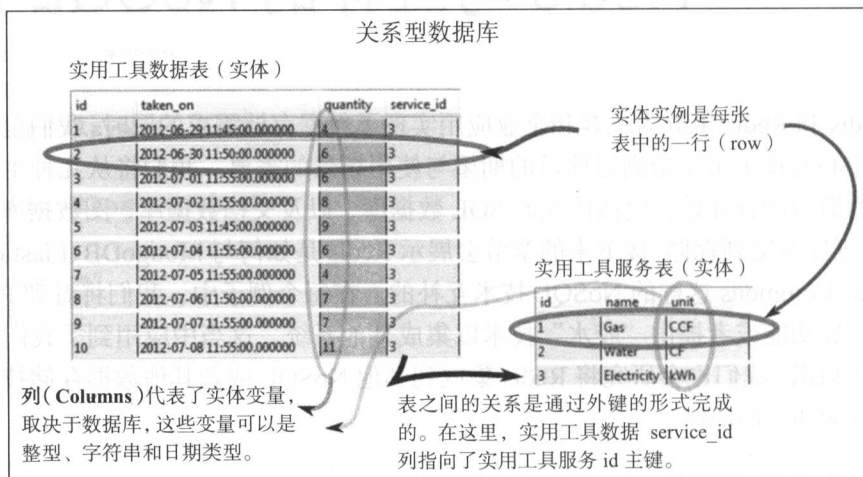
在 Redis 和 Redis 集群为顾客和企业应用实现了数据存储需求的同时,我们也需要其他以数据为中心的技术来完全满足项目的期望与使用场景的需要。我们将从几种主要的数据存储技术的简短调查开始,包括传统的 SQL 数据库,以及文档数据库、图数据库、搜索索引、键值数据库及宽列存储。接下来的章节会展示 Redis 是如何与 MongoDB、ElasticSearch, 以及 Fedora Commons 这样的 NoSQL 技术互补的。在每个例子中,我们将看到 Redis 是如何被用于扩展功能或者提供“胶水”技术以集成其他系统。这当中应用到了我们在之前章节中习得的知识。我们也会研究将 Redis 集成到其他 NoSQL 或者其他数据存储技术所带来的成本与可能遇到的障碍。

NoSQL 技术的繁荣

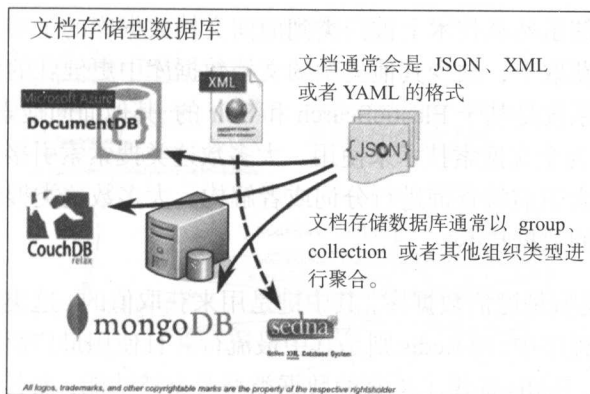
在过去的 10 年里,大量的数据存储技术崭露头角,为数据密集型的应用带来更多选择。基于数据的存储,数据的操作及数据的返回,可以大致将这些存储技术进行归类,并对它们的流程序进行跟踪及排名。Redis 作为一种键值存储技术,在使用性和流行程度上均有所改进,目前排在数据存储技术的前 10 位,详情请参考附录中来源的第 7 章:Redis 与互补的 NoSQL 技术的第一要点。

关系型数据库,特别是那些支持 SQL 的关系型数据库,是历史最悠久、最流行的数据存储技术。从像 Oracle 和 Microsoft 这样的大型企业级关系型数据库管理系统(DBMS),到流行甚广的 MySQL(现在被 Oracle 收购)和 Postgres 开源系统,这些数据库已经逐渐成为大多数企业混合数据存储技术的一部分。对于小型组织来说,这些 DBMS 中的一种可能就是唯一可用的数据存储技术。这些数据库通常是大型客户管理系统、会计、库存或者其

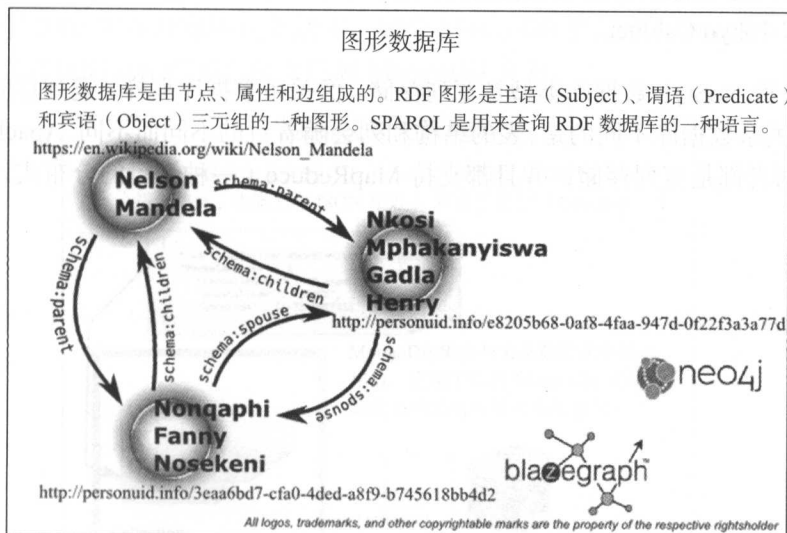
他企业级技术的一部分。在关系型数据库中，数据以表的方式组织，每张表通常代表了单一实体，其中的每行记录代表了实体的实例，而列则包含了实体的变量，可以设置是否是必须的。每一行包含了唯一 ID 或者由单独的列组成的组合键。表之间通过使用外键的方式进行关联，其中一张表的主键存储在另一张表的列中。同时，应用程序或者数据库系统通过这些外键关系将表进行关联 (join) 获取数据。如果单独列中的数据与其他表中的没有重复，那么该关系型数据库就是规范化的。构建和查询关系型数据库最流行的方式是使用声明式语言 SQL (结构化查询语言)。对大多数数据库来说，这已成为事实上的标准，它们对 SQL 的实现会有差异，有时也会对 SQL 进行扩展。



面向文档的数据库，相对于关系型数据库而言，是基于管理和操作半结构化的数据结构。这种数据结构被（非正式的）称作文档。在下一节中，我们将研究最流行的文档型存储之一的 MongoDB。其他面向文档的数据库还包括 CouchDB、Sedna、微软的 DocumentDB、Jackrabbit、IBM 的 Informix 及 MarkLogic。只要文档有结构类型，那么文档的格式可以是 XML、JSON 和 YAML 中的一种。XML 和 JSON 这两种格式是非常流行的选择，已经演变成广泛的面向文档的数据库中独特的细分市场。撇开格式不谈，文档存储通常会依赖于数据的结构提供查询或者获取文档的方法。每份文档都拥有唯一 ID，用于代表文档自身。许多最流行的面向文档的数据库都有自定义查询语言用于获取文档，这些查询的性能随着文档存储的不同而不同。



图形数据库使用节点、边及属性存储和操作数据。图形数据库中的节点代表了一个对象或者实体，包含了一个或多个属性，而边则代表数据库中不同节点间的连接或者关系。图形数据库允许通过基本的关联逻辑在不同节点间的关系上做更简单的推断。图形数据库没有形式化的模式，这导致了允许异构数据源的更简单的集成。一些知名的图形数据库包括 AllegroGraph、Blazegraph（原先是 BigData）、InfiniteGraph、Neo4j、OpenLink Virtuoso、Oracle 的 NoSQL 和 patial 产品、OrientDB 及 Stardog。用于获取和查询图形数据库的最流行的方法是通过图形数据库对 SPARQL（SPARQL 协议和 RDF 查询语言）的实现和支持。SPARQL 允许用户构造复杂的查询来获取节点及操作属性和节点间的边。

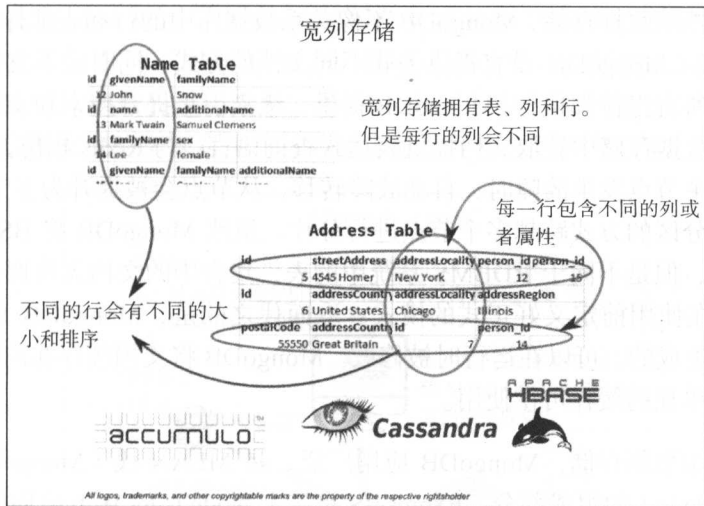


全文搜索数据存储虽然从技术上被归类到面向文档数据库的子集，不过它聚焦在基于用户查询结果的快速获取上，缺少其他类型的文档数据库中更健壮的操作和管理功能。一些流行的基于搜索的系统是基于 ElasticSearch 和 Solr 的。其他面向搜索的系统，例如 Sphinx 和 Xapian，也可以作为全文搜索技术来使用。大多数这类搜索索引接受非形式化的用户查询，在执行查询搜索索引前将查询进行分词或者解构。大多数这类搜索数据库都缺乏事务性或者其他更为健壮的数据技术。

下一个 NoSQL 类型是键值数据库，其中键是用来获取值的。这类数据库正越来越多地被使用在广泛的应用程序中，而 Redis 则是其中最流行并且使用最广泛的键值数据库范例。作为贯穿本书的主题，Redis 提供了丰富的数据类型及支持功能，而其他键值数据库仅仅实现了关联数组，通过一个键获取一个不透明¹的值，然后在发起调用的客户端代码里使用。键值数据库可以被分为不同的类别，这其中只有少数几种可以横跨多个类别。对于那些提供最终一致性保证的键值数据库来说，如果没有发生更新，那么分布式数据库中的所有节点会提供最近更新的值。提供最终一致性的数据库有 Amazon 的 DynamoDB、Oracle 的 NoSQL 数据库及 Riak。第二种类型的键值数据库提供通过键或者值对数据进行排序的功能，有序数据库的例子有 Berkeley DB、HyperDex、InfinityDB 及 LMDB。第三种键值数据库是内存模式数据库。其中 Redis 最广为人知，还有其他 RAM 数据库，包括 Aerospike、Oracle 的 Coherence、memcached 及 OpenLink Virtuoso。最后一种键值数据库类型是磁盘模式数据库。数据从固态硬盘或者传统磁盘进行读写操作。磁盘模式数据库包括 BigTable、Hibari、LevelDB 和 Tokyo Cabinet。

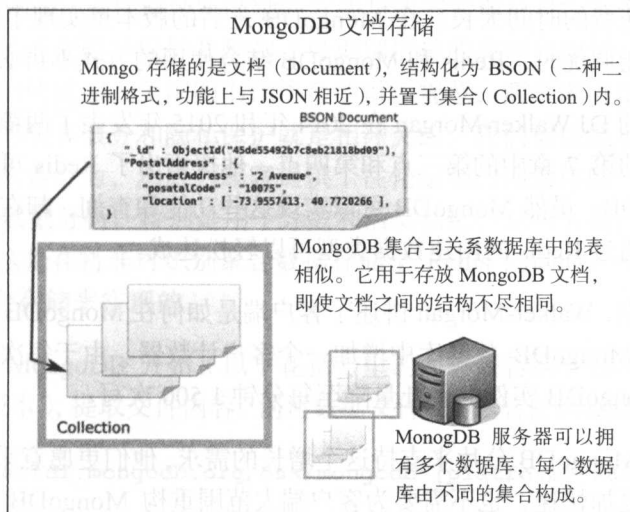
最后一种 NoSQL 数据库类型是宽列存储。像关系数据库那样，宽列存储也有表格、行及列。但与关系数据库不同的是，表的结构和列会随着行的不同而不同。Apache 的 Cassandra 和 HBase 两者都是宽列存储，并且都支持 MapReduce（一种流行的分布式计算方法）。

1 译者注：相对于 Redis 提供的数据类型，客户端可以直接获取想要的最终值，而其他键值数据库获取到值之后，还需要再进行解构操作，才能获取想要的值。



Redis 作为 MongoDB 的分析补充

作为最流行的 NoSQL 数据库类型之一，MongoDB 被归类为文档存储，数据围绕着操作和搜索进行，组织成基于 JONS 的文档变体，被称为 BSON（二进制序列化对象标记的简写）。MongoDB 中的 Mongo 是从单词 humongous 中抽取出来的，在 2007 年由 MongoDB 公司发起，并于 2009 年在开源协议下发布。如今 MongoDB 仍然由 MongoDB 公司赞助并开发。客户可以选择向公司购买企业支持和 MongoDB 托管。



作为面向文档的数据存储, MongoDB 不像关系数据库中的行和表那样,而是将对象存储为 BSON 格式。MongoDB 没有提供关联不同文档的方式,同时也不支持多个文档间的原子事务来保证所有操作都执行。MongoDB 提供二级索引来提升搜索和文档级别的原子操作,并且针对从数据存储中获取文档提供表达式查询语言。与 Redis 相仿, MongoDB 提供了主从复制,当主节点发生故障时,自动故障转移,从节点会被提升为主节点。MongoDB 也支持通过范围分区的方式跨越多个节点进行分片。虽然 MongoDB 将 BSON 文档通过集合进行分组管理,但是不同于 RDBMS 系统中的表,集合中的文档无须拥有相同的结构。MongoDB 无须在使用前定义好正式的模式。取而代之的是, MongoDB 的动态模式是从 BSON 文档结构生成的,可以在运行时被修改。MongoDB 将文档缓存在内存中,但是对于应用程序来说没有单独的缓存可供使用。

作为一个通用数据存储, MongoDB 应用广泛,是 MEAN 栈 (MongoDB、ExpressJS、AngularJS 和 Node.js) 的组成部分。ExpressJS 是一个 Node.js 的 Web 应用框架, AngularJS 提供了动态客户端作为 Web 前端,而后端由 MongoDB 满足应用程序的存储需求。MEAN 人气暴涨的原因在于不少团队对于提供端到端的 JavaScript 应用程序十分感兴趣。他们从 ExpressJS 和 Node.js 那里存储并获取 JSON 数据,然后将其传输到使用 AngularJS 编写的富客户端用 HTML 做展现。

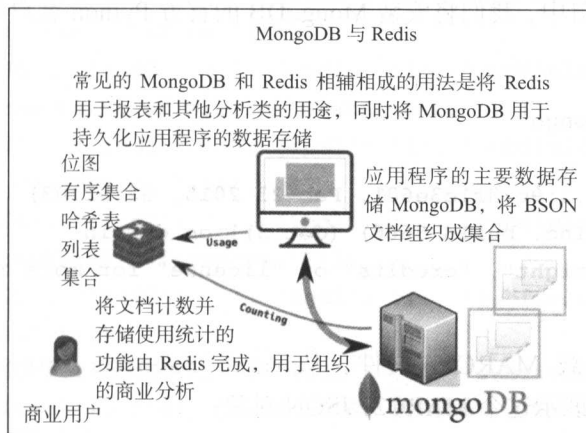
在 MongoDB 实现生命周期 (TTL) 功能之前,通常会采用 Redis 触发 MongoDB 数据库中的文档和集合的自动删除。2012 年, Cody Powell 在博客中提到 (详情请参考附录中来源的第 7 章中的第二点),他所使用的 MongoDB 是时间密集型的,专门用于移动游戏应用程序推荐引擎提供后台数据存储。虽然从 MongoDB 读取数据非常快速,但是对于他的需求来说插入数据花费的时间太长。在 MongoDB 之后的版本里实现了集合的 TTL,因此当为应用程序做性能调优时, Redis 和 MongoDB 结合使用的方式不再必要。

Compose 公司的 DJ Walker-Morgan 在 2014 年和 2015 年发表了两篇文章,详情请分别参考附录中来源中的第 7 章中的第三点和第四点。他探索出了 Redis 可以通过转换功能和查询来辅助 MongoDB。虽然 MongoDB 可以实现这些功能和查询,却在内存消耗和时间上非常昂贵。不过,通过 Redis 的内建数据类型可以轻松达成。

在第一篇文章中, Walker-Morgan 讲述了客户端是如何在 MongoDB 应用程序中遭遇瓶颈的。当时需要在 MongoDB 数据库中增加一个客户计数器。由于每次插入需要向磁盘写一次数据,单一 MongoDB 实例的吞吐量降至每分钟 1 500 次写。

相比创建多个 MongoDB 分片来支持这个增长的需求,他们更愿意采用 Redis 增加客户计数器,这样做会更加快速,也不需要为客户端大范围重构 MongoDB 数据存储。在第二

篇文章中, Walker-Morgan 概述了 Redis 直接操作数据的能力是如何成为 MongoDB 强有力的补充的。他举的第一个例子回到了 Redis 对于快速增加和存储整数的能力及如何通过 Redis 键模式约定为这些增长进行建模。在第二个例子中, 他讲述了如何采用 Redis 中 HyperLogLog 数据类型的实现对用户计数的设置, 并在之后进行获取。



为了试验 Redis 是如何辅助并扩展 MongoDB 的, 我们将建模使用场景。在该场景中, 我们将 MARC21 记录序列化为 JSON, 并存储为单一 MongoDB 实例。首先, 我们使用 Redis 实例存储记录使用计数器, 使用 Redis 的 bit-string 记录每天的客户数, 以便之后用于创建每隔任意时间周期内的总量指标。

通过将指标分析从数据存储中分离, 我们在该案例中使用 Redis 进行指标分析, 采用 MongoDB 进行数据存储。这种设计所鼓励的方法与图书管理员对顾客隐私关怀的精神是一致的。图书馆目录的道德设计保护并限制了可识别信息的类型和数量, 以保护顾客的信息查询行为的隐私。即使被编入美国图书馆协会的道德规范, 两个图书馆理念之间也经常有冲突。规范中规定的顾客服务和隐私保护就是相冲突的。我们提供低水平的服务, 因为我们不能追踪和分析顾客行为, 然后为顾客提供个性化定制的搜索体验。在此场景中, 通过使用 Redis 做用户数据分析, 特别是用于计数和活动指标, 意味着我们能将私密信息滥用的情况降至最低。这是在将非可识别聚合数据存储至 Redis 的同时, 将图书馆系统永久保存至 MongoDB 后台存储来实现的。

我们将从下载 MongoDB 开始 (以下花括号里的内容请替换成你所使用的平台及当前 MongoDB 的稳定版本), 提取文件内容, 创建数据文件夹, 然后使用默认设置运行 Mongo:

```
$ wget https://fastdl.mongodb.org/osx/mongodb-{platform}-x86_64-
{version}.tgz
```

```
$ tar -xvf mongodb-{platform}-x86_64-{version}.tgz mongodb
$ cd mongodb
$ mkdir -p data/db
$ ./bin/mongod --dbpath {path-to-data-db}/data/db
```

在第二个终端窗口中，我们将安装 MongoDB 的官方 Python 客户端，并启动 Python 实例：

```
$ pip3 install pymongo
$ python3
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 23 2015, 02:52:03)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

现在，我们将加载 MARC21 文件，从 `marc_records` Python 列表中获取一条 MARC21 记录，然后展示这个 MARC21 JSON 对象：

```
>>> import json, pymarc
>>> marc_records = [r for r in pymarc.MARCReader(open("/var/tmp/samplemarc.mrc",
"rb"), to_unicode=True)]
>>> sample_record = marc_records.pop(67)
>>> marc_json = json.loads(sample_record.as_json())
>>> marc_json
{'fields': [{'001': '4356682'}, {'008': 'eng'},
             {'035': {'ind2': ' ', 'ind1': ' ', 'subfields': [{'a':
'.b10019947'}, {'b': 'tbp'}, {'c': '-'}]}},
             {'035': {'ind2': ' ', 'ind1': ' ', 'subfields': [{'a':
'(CoCC)102429'}]}}, {'040': {'ind2': ' ', 'ind1': ' ', 'subfields':
[{'a': 'MUU'}, {'c': 'MUU'}, {'d': 'm.c'}]}},
             {'049': {'ind2': ' ', 'ind1': ' ', 'subfields': [{'a':
'COCA'}]}},
             {'090': {'ind2': ' ', 'ind1': ' ', 'subfields': [{'a':
'PR2825.A2 B7 1967'}]}},
             {'100': {'ind2': ' ', 'ind1': '1', 'subfields': [{'a':
'Shakespeare, William,'}, {'d': '1564-1616.'}]}}},
             {'245': {'ind2': '4', 'ind1': '1', 'subfields': [{'a': 'The
merchant of Venice /'}, {'c': 'edited by John Russell Brown.'}]}}},
```

```

    {'250': {'ind2': ' ', 'ind1': ' ', 'subfields': [{'a': '7th
ed., rev.'}]}}},
    {'260': {'ind2': ' ', 'ind1': ' ', 'subfields': [{'a':
'London :'}, {'b': 'Methuen,'}, {'c': '1964, 1967 printing.'}]}}},
    {'300': {'ind2': ' ', 'ind1': ' ', 'subfields': [{'a':
'lviii, 174 p. ;'}, {'c': '22 cm.'}]}}},
    {'490': {'ind2': ' ', 'ind1': '1', 'subfields': [{'a': 'The
Arden edition of the works of William Shakespeare.'}]}}},
    {'490': {'ind2': ' ', 'ind1': '1', 'subfields': [{'a': 'The
Arden Shakespeare Paperbacks.'}]}}},
    {'504': {'ind2': ' ', 'ind1': ' ', 'subfields': [{'a':
'Includes bibliographical references.'}]}}},
    {'700': {'ind2': ' ', 'ind1': '1', 'subfields': [{'a':
'Brown, John Russell.'}]}}},
    {'800': {'ind2': ' ', 'ind1': '1', 'subfields': [{'a':
'Shakespeare, William,'}, {'d': '1564-1616.'}, {'t': 'Works.'}, {'f':
'1954.'}]}}},
    {'800': {'ind2': ' ', 'ind1': '1', 'subfields': [{'a':
'Shakespeare, William,'}, {'d': '1564-1616.'}, {'t': 'Works.'}, {'f':
'1954.'}, {'s': 'Paperbacks.'}]}}},
    {'830': {'ind2': '0', 'ind1': ' ', 'subfields': [{'a':
'Arden edition of the works of William Shakespeare.'}]}}},
    {'907': {'ind2': ' ', 'ind1': ' ', 'subfields': [{'a':
'.b10019947'}]}},
    {'902': {'ind2': ' ', 'ind1': ' ', 'subfields': [{'a':
'150104'}]}},
    {'999': {'ind2': ' ', 'ind1': ' ', 'subfields': [{'b': '2'},
{'c': '940803'}, {'d': 'm'}, {'e': 'a'}, {'f': '-'}, {'g': '4'}]}},
    {'994': {'ind2': ' ', 'ind1': ' ', 'subfields': [{'a':
'tbp'}]}},
    {'945': {'ind2': ' ', 'ind1': ' ', 'subfields': [{'a':
'PR2825.A2 B7 1967'}, {'g': '1'}, {'i': '33027001268287'}, {'j': '0'},
{'l': 'tbp '}, {'h': '0'}, {'o': '-'}, {'p': '$0.00'}, {'r': '-'}, {'s':
'-'}, {'t': '1'}, {'u': '10'}, {'v': '0'}, {'w': '0'}, {'x': '1'}, {'y':
'.i10024165'}, {'z': '940804'}]}},
    'leader': '01144nam 2200313 4500'

```

在将 `marc_json` 插入 Mongo 数据库前, 我们需要先引入 Python 的模块 `pymongo`, 并实例化 Mongo 客户端:

```
>>> from pymongo import MongoClient
>>> client = MongoClient()
```

为了将 `marc_json` 作为 MongoDB 文档插入数据库, 我们首先创建一个 Mongo 数据库和集合, 然后调用 `insert_one` 方法将 `marc_json` 插入 MongoDB 数据库并存储结果 ID:

```
>>> marc_db = client.marc_db
>>> marc_collection = marc_db.marc_collection
>>> sample_record_id = marc_collection.insert_one(marc_json).inserted_id
>>> sample_record_id
ObjectId('55e9958f0f55c501f6802edf')
```

在实现 Redis 的使用计数器和人数统计时, 我们将实例化 Python 的 Redis 客户端并连接至单个 Redis 实例, 其运行在默认的 `localhost:6379` 上:

```
>>> import redis
>>> marc_redis = redis.StrictRedis()
```

我们首先将 `ObjectId` 插入 Redis 有序集合中, 并将权重设置为递增插入后的偏移量, 之后会用来记录 MARC 记录的使用情况:

```
>>> offset = marc_redis.incr("insertion-offset")
>>> marc_redis.zadd("marc-insertion", offset, str(sample_record_id))
1
```

现在, 我们将使用 JSON 版本的 MARC 记录填充 MongoDB, 同时递增 `insertion-offset` 计数器并将 MongoDB Object 添加到有序集合 `marc-insertion` 中, 并用作 `insertion-offset` 的分数 (score)。之前的逻辑由 Python 的函数 `process_records` 实现, 存放于 Python 代码文件 `marc_example.py` 中。它遵循 Apache2 协议并伴随着本章, 可以在本书的网站或者 GitHub 库上下载。我们将从 `marc_example` 引入 `process_records`, 并将我们的样例 MARC 记录传入:

```
>>> import marc_example
>>> marc_example.process_records(marc_records)
```

在运行我们的测试记录集之后, 我们应该能够从 Redis 实例和 MongoDB 数据存储中得到人数总计:

```
>>> marc_collection.count()
17145
>>> marc_redis.get('insertion-offset')
b'17145'
```

那么，为了测试我们对使用 Redis 实时分析技术能够更快的获取计数这一假设，我们将使用 Python 的 `timeit` 代码模块测量代码片段的执行时间。在该测试中，从 MongoDB 集合中获取 MARC JSON 记录的总数与从 `marc_redis` 实例中获取 `insertion-offset` 的当前值，将分别默认运行一百万次²。

```
>>> marc_collection_timeit = timeit.timeit(stmt=marc_collection.count)
>>> marc_collection_timeit
167.59296235199963
```

针对 Redis 的测试需要运行设置语句，创建 Redis Python 客户端 `marc_redis`，然后对 `insertion-offset` 运行 GET 命令：

```
>>> redis_get_timeit = timeit.timeit(stmt='marc_redis.get("insertion-
offset")',
    setup="import redis; marc_redis=redis.StrictRedis()")
>>> redis_get_timeit
66.52741511400018
```

测试示例有点粗糙，没有考虑 Redis 或者 MongoDB 的 Python 代码客户端的任何延迟。运行 GET 命令和对 MongoDB 集合 `count` 方法的调用，这两者的差异体现在 Redis 分析返回人数总计要比 MongoDB 快两倍。GET 命令的时间复杂度为 $O(1)$ ，相比 MongoDB 测试来说拥有更好的表现。我们也可以测试另一个不同的 Redis 命令 `ZCARD`，以 $O(1)$ 的时间复杂度获取人数统计：

```
>>> redis_zcard_timeit = timeit.timeit(stmt='marc_redis.zcard("marc-
insertion")',
    setup="import redis; marc_redis=redis.StrictRedis()")
>>> redis_zcard_timeit
66.67229959900033
```

正如我们预期的那样，`ZCARD` 和 `GET` 这两个命令花费了大约相同的时间，获取

2 译者注：Python 的 `timeit` 的参数中包含了 `number` 参数，即运行次数，默认值 1 000 000，这里使用了该默认值。

1 000 000 次调用花费 66s 的时间。基于以上结果，我们可以调整 Redis 分析键模式，使用单个有序集合来满足要求，取代递增的偏移量。



有一个有趣的练习留给各位读者。请分别采用 Ruby、Node.js 及 Java 版本的 MongoDB 和 Redis 客户端重复上述实验，以测量 MongoDB 和 Redis 客户端性能。

即便只有 20 000 条 MARC21 记录的小型样本，我们也能够进入下一个实验，用来追踪 MongoDB 或者 Redis 目录中 MARC 记录的每日使用数。为了使用 MongoDB 追踪这类使用情况，我们需要创建一个新的集合 `marc_usage`，用于存放包含 BSON 时间戳和 MARC21 对象 ID 的 JSON 对象。我们将使用 MongoDB 查询获取所有 24 小时周期内的使用情况。我们的 Redis 解决方案将使用 MongoDB ID 偏移量。首先在有序集合 `marc-insertion` 中通过 `ZRANK` 命令获取分值，然后将那一天的位串中对应偏移量的位进行设置。在我们的例子中，如果 MARC 记录在借阅事件中被使用了（一天只会发生一次），那么我们将它定义为使用情况。

在 `marc_example` 代码文件中，另一个名为 `add_mongo_daily_usage` 的函数，创建并写入一份由时间戳及指向 `object_id` 的引用所组成的使用情况文档：

```
def add_mongo_daily_usage(object_id, date):
    usage_collection = MARC_USAGE.usage_collection
    usage_document = { "datetime": date.isoformat(),
                       "marc-id": str(object_id) }
    return usage_collection.insert_one(usage_document).inserted_id
```

我们使用一个叫作 `add_redis_daily_usage` 的函数对基于 `object_id` 的位串进行设置：

```
def add_redis_daily_usage(offset, date):
    usage_key = date.strftime("%Y-%m-%d")
    MARC_REDIS.setbit(usage_key, offset, 1)
```

为了模拟 90 天的使用量，我们使用 500 至 1000 的随机偏移地址用于产生日常使用量，并通过 `add_mongo_daily_usage` 函数生成 MongoDB 使用量文档，同时调用 `add_redis_daily_usage` 函数设置 Redis 分析实例相应的字符串位。我们将运行 `marc_example` 代码文件中的 `run_usage_simulation` 函数生成用于运行性能测试的

测试数据。我们将从 MongoDB 和 Redis 分析实例中获取日常使用量。

```
def run_usage_simulation(seed_seconds, runs=90):
    seconds_in_day = 60*60*24
    max_records = int(MARC_REDIS.get('insertion-offset'))
    for day in range(runs):
        timestamp = datetime.datetime.utcnow().timestamp(
            seconds_in_day*day + seed_seconds)
        daily_usage = random.randint(500, 1000)
        for use in range(daily_usage):
            offset = random.randint(1, max_records)
            result = MARC_REDIS.zrange('marc-insertion', offset, offset)
            if len(result) < 1:
                continue
            object_id = result[0]
            add_mongo_daily_usage(object_id, timestamp)
            add_redis_daily_usage(offset, timestamp)
```

运行这份模拟结果给了我们两种方式来查明 90 天时间周期内的使用量。将使用量文档存储于 `usage_collection` 意味着我们能够通过在 Python 终端会话上执行 `usage_collection.count` 方法获得粗略的统计：

```
>>> usage_collection.count()
66712
```

在填充了模拟 90 天周期使用量并将结果存储于 MongoDB 和 Redis 之后，我们将比较 MongoDB 和 Redis 两者间日常查询的性能。首先使用接下来的 MongoDB 查询语句获取计数：

```
>>> usage_collection.count({ "datetime": { '$lt': '2015-11-
06T00:00:00.0', '$gt': '2015-11-05T00:00:00.000Z' } })
681
```

在使用 Redis 计算同样的每日使用情况计数时，需要在那一天上使用 Redis 命令 `BITCOUNT`：

```
>>> marc_redis.bitcount("2015-11-05")
671
```


为什么对 10 月 15 日这一天来说，MongoDB 和 Redis 的位计数会有 10 个单位的差异呢？一种可能的情况是单份 MARC 文档在一天内被使用了多次，每次发生都对应一份使用文档。我们可以在之后通过再次运行查询来调查这一差异。不过现在我们来迭代所有的记录并检查是否确有 10 条重复记录：

```
>>> nov5_ids = {}
>>> duplicates = 0
>>> for doc in daily_usage:
    marc_id = doc.get('marc-id')
    if marc_id in nov5_ids:
        duplicates += 1
    else:
        nov5_ids[marc_id] = 1
>>> duplicates
10
```

这个简单的测试确认了在我们的模拟中，问题发生的原因在于一天当中处理了多次借阅。我们可以通过当前实现的不同方式解决这一数据问题。改进 MongoDB 查询功能增加过滤重复记录是其中一种方案。不过，这一方案为解决方案引入了额外的代码复杂性，需要在产品环境中进行维护操作。一个更好的方案是改进 `run_usage_simulation`，用以下代码替换 `random.randint` 函数：

```
daily_usage = random.randint(500, 1000)
offsets = random.sample(range(1, max_records), daily_usage)
for offset in offsets:
    result = MARC_REDIS.zrange(
        'marc-insertion',
        offset,
        offset)
```

为了重新测试，我们将从空的 MongoDB 和 Redis 实例开始，并重新执行 `run_usage_simulation`，这次我们从 MongoDB 和 Redis 中获取了 10 月 5 日那天的同样的日常计数数据：

```
>>> usage_collection.count({"datetime": {"$lt": "2015-11-06T00:00:00.0", "$gt": "2015-11-05T00:00:00.000Z"}})
577
>>> marc_redis.bitcount("2015-11-05")
```

577

现在，我们将再次使用 `timeit` 运行 10 000 次试验以获取每日计数的平均性能，首先从 Redis 开始测试：

```
>>> redis_setup = """import redis
marc_redis = redis.StrictRedis()"""
redis_daily_count_test = timeit.timeit(stmt= """marc_redis.
bitcount("2015-11-05")""",
    setup=redis_setup,
    number=10000)
>>> redis_daily_count_test
0.6927584460008802
```

现在，我们对 MongoDB 做同样的测试：

```
>>> mongodb_setup = """from pymongo import MongoClient
client = MongoClient()
usage_collection = client.marc_usage.usage_collection"""
>>> mongo_stmt = """usage_collection.count({ "datetime": { '$lt': '2015-
11-06T00:00:00.0', '$gt': '2015-11-05T00:00:00.000Z'}})"""
>>> mongodb_daily_count_test = timeit.timeit(
    stmt=mongo_stmt,
    setup=mongodb_setup,
    number=10000)
>>> mongodb_daily_count_test
274.93577796200043
```

即便使用了相对较少的试验数量，获取 Redis 和 MongoDB 日常使用量在时间上的差异是显而易见的，Redis 获取 10 000 次使用情况需要不到 1s，而 MongoDB 则需要 274s。再次提醒注意，你的测试结果会因为硬件和软件设置的不同而不同，而且对于 MongoDB 来说，这个例子可能在构造文档和查询上有优化的空间，可以提升 MongoDB 的性能。

通过使用 Redis 命令 `BITOP`，我们将进一步计算在期间内总的记录使用情况，通过连接到 Redis 分析实例的 `redis-cli` 会话，运行下列命令：

```
127.0.0.1:6379> BITOP OR "2015:christmas-week" "2015-12-19" "2015-12-20"
"2015-12-21" "2015-12-22" "2015-12-23" "2015-12-24" "2015-12-25"
(integer) 2143
```

```
127.0.0.1:6379> BITCOUNT "2015:christmas-week"  
(integer) 4710
```

在我们的 Python MongoDB 客户端上运行同样的命令，结果如下：

```
>>> usage_collection.count({ "datetime": { "$gt":  
"2015-12-19T00:00:00.000Z", '$lt': '2015-12-26T00:00:00.0', }})  
5371
```

为什么我们获取到的结果不一样呢？这是由 BITOP OR 操作的性质决定的，它并没有实际上执行 bitcount 操作，而是计算了所有位图的联合结果，任何被设置成 1 的位只被计 1 次，即便某几天有多次的情况。换句话说，Redis 的键 `2015:christmas-week` 存储了这段时间内所有唯一的使用情况，而不是总的访问量。我们可以通过循环每一天，获取每天的 BITCOUNT 值，然后累加来证明这一点：

```
>>> christmas_count = 0  
>>> for day in range(19, 26):  
    key = "2015-12-{}".format(day)  
    count = marc_redis.bitcount(key)  
    christmas_count += count  
    print(key, count, christmas_count)
```

```
2015-12-19 656 656  
2015-12-20 784 1440  
2015-12-21 745 2185  
2015-12-22 891 3076  
2015-12-23 825 3901  
2015-12-24 616 4517  
2015-12-25 854 5371
```

通过这个简单的示例，我们可以看到 Redis 是如何辅助 MongoDB 数据存储的：在 MongoDB 上某些耗时又昂贵的任务，Redis 可以完成得更高效。并不是所有数据都适用于 JSON 文档，我们已经通过 Python 的 `timeit` 模块验证了这一点。我们采用 Redis 处理应用程序中的分析模块，而采用 MongoDB 作为主要的数据存储，这将有助于计数和其他报表任务的执行速度，同时通过结构合理的 BSON 格式简化了 MongoDB 的数据持久化。

Redis 作为 Elasticsearch 的预处理组件

ElasticSearch 为 Lucene 增加了基于 JSON 的前端处理。Lucene 是一个开源的企业级搜索引擎, 由 Apache 基金会发起, 同时也是另一个流行的搜索技术 Solr 的核心。ElasticSearch 将 JSON 文档建立为 Lucene 索引, 并使用自定义基于 JSON 的 DSL (领域特定语言) 查询搜索索引。ElasticSearch 使用分片和集群技术对搜索进行扩展, 以包含大型数据集。ElasticSearch 为众多知名网站如 Netflix、纽约时代周刊、思科、eBay 和高盛提供搜索支持。

ElasticSearch 的主要发起者是营利公司 Elastic.co。该公司也同时支持许多其他补充或者构建于 ElasticSearch 之上的搜索索引。例如 Logstash, 它是一种日志收集器, 将对日志建立索引到 ElasticSearch。还有 ElasticSearch 的虚拟化工具 Kibana。本章会着重讲解上述三者和 Redis 的使用。



在 BIBCAT 中使用 Redis 和 Elasticsearch

在为国会图书馆设计和开发一个关联数据数目搜索和展示系统 (以下将书目分类缩写为 BIBCAT) 期间, 采用 Redis 为 BIBFRAME 1.0 RDF 图进行初始重复数据删除的预处理。虽然 ElasticSearch 能够通过对 BIBFRAME authorizedAccessPoint 三元组进行 ElasticSearch term 查询以轻松应对重复数据删除, 但如果使用 Redis 作为替代, 我们就能使用关联数据片段服务器更加精确地匹配三元模式。关联数据片段服务器又快又能简化匹配

逻辑。

如果采用关联数据片段服务器,就允许匹配算法与方法的可替代性测试,例如 OCLC (联机计算机图书馆中心),网站地址 <https://viaf.org>。该网站聚集了官方国家图书馆声明(称作授权),其中关系到作家、作曲家、艺术家、摄影师、插画师、组织、学科及其他拥有与创意作品有关的可识别角色或者关系的实体。

我们这就开始,先从 <https://www.elastic.com/> 上获取最新的 Elasticsearch TAR 文件,解压并运行:

```
$ wget https://download.elastic.co/elasticsearch/elasticsearch/
elasticsearch-1.7.1.tar.gz
$ tar xvf elasticsearch-1.7.1.tar.gz
$ mv elasticsearch-1.7.1
$ ./elasticsearch/bin/elasticsearch
```

虽然在实际 BIBFRAME 数据存储中,我们针对 BIBFRAME 词汇加载自定义的映射和配置,但在我们的示例中,我们将为 Elasticsearch 使用默认的映射和配置。就该示例来说,我们将从两个样例 MARC21 文件中创建 BIBFRAME RDF 图开始,第一份文件是由与简·奥斯丁的《傲慢与偏见》相关的 MARC21 记录组成,第二份示例则是由与赫尔曼·梅尔维尔的《白鲸》相关的 MARC21 记录组成:

```
>>> import pymarc
>>> pride_and_prejudice = [r for r in pymarc.MARCReader(open("/var/tmp/
automatic-bibframe-classification/ColoradoCollege/pride-and-prejudice.
mrc", "br+"), to_unicode=True)]
>>> moby_dick = [r for r in
pymarc.MARCReader(open("/var/tmp/automaticbibframe-classification/ColoradoCo
llege/moby-dick.mrc", "br+"), to_
unicode=True)]
>>> len(pride_and_prejudice)
30
>>> len(moby_dick)
22
```

接下来,我们将为这 52 条 MARC 记录运行 `convert2bibframe` 函数,该函数将使用来自国会图书馆的 `marc2bibframe` 项目和套接字服务器函数 `xquery_socket`。`xquery_socket` 使用 Jython 将转换器和轻量级套接字服务器包装在一起。Jython 是一个

将 Python 代码运行在 JVM 上的 Python 项目。

```
>>> def convert2bibframe(record):
return xquery_socket(pymarc.record_to_xml(record, namespace=True))
>>> pp_bibframe = [convert2bibframe(r) for r in pride_and_prejudice]
```

当转换白鲸 MARC 记录时, 从 convert2bibframe XQuery 函数可以看到其中的第 9 条记录发生了错误。我们从有问题的记录中发现它缺失了 BIBFRAME 转换流程所需的 MARC 001 字段。当我们为其添加上 001 字段桩 (stub) 时, 转换就成功了:

```
>>> moby_dick[9].add_field(pymarc.Field('001', data='1415005'))
>>> md_bibframe = [convert2bibframe(r) for r in moby_dick]
>>> len(md_bibframe), len(pp_bibframe)
(22, 30)
```

我们的三元组总数可以通过使用 Python 的 sum 方法进行计算, 它会对每个图分别进行求和计算, 返回《傲慢与偏见》和《白鲸》各自的三元组总数:

```
>>> (sum(len(g) for g in pp_bibframe), sum(len(g) for g in md_bibframe))
(7116, 3113)
```

这 52 条 MARC21 记录产生了 10 229 个三元组。每个三元组将作为输入传入 Redis 预处理器。Redis 预处理器会删除重复的数据对象, 并在创建 JSON 内容添加到 Elasticsearch 进行索引之前将三元组添加到 Redis 缓存中。举例来说, 在我们的《傲慢与偏见》样例集合中, 我们期望看到简·奥斯汀作为作者和主语, 并且在进行 Elasticsearch 索引前, 我们的目录应当将所有分隔的简·奥斯汀 BIBFRAME 个人实体汇聚成单一实体。我们先创建一条 SPARQL 查询运行每个《傲慢与偏见》BIBFRAME 图:

```
>>> sparql_query = """PREFIX bf: <http://bibframe.org/vocab/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?sub ?pt
WHERE {
    ?sub rdf:type bf:Person .
    ?sub bf:authorizedAccessPoint ?pt
}"""
```

如果只是想看看 BIBFRAME Person 的 RDF 主语, 我们将只展示第一个图的 Person:

```
>>> for row in pp_bibframe[0].query(sparql_query):
```

```
print(row[1])
```

Chapter 7

[215]

Ehle, Jennifer, 1969-
 Chancellor, Anna.
 Bamber, David.
 Steadman, Alison, 1946-
 Sawalha, Julia, 1968-
 Whitrow, Benjamin.
 Langton, Simon.
 Bonham-Carter, Crispin.
 Harker, Susannah.
 Firth, Colin, 1960-
 Austen, Jane, 1775-1817.
 Austen, Jane, 1775-1817--Film adaptations.

如果我们对最后一个图进行一样的查询，会期望看到至少另外一个 `bf:authorizedAccessPoint` 的主语为 `Austen, Jane, 1775-1817`:

```
>>> for row in pp_bibframe[-1].query(sparql_query):
    print(row[1])
```

```
Cronin, Richard, 1949-
McMillan, Dorothy, 1943-
Austen, Jane, 1775-1817.
```

随着我们将 RDF 图添加到 Redis 缓存，我们将检查主语是否含有 `BIBFRAME` `authorizedAccessPoint`，以及该属性是否匹配之前已存在的主语。如果存在匹配，那么原来的主语会被新进的主语替换掉，并且对原来的主语来说，不存在的三元组将被添加到关联数据片段服务器。这些三元组中包含了之前存在的主语、全新的谓语和宾语：

```
def dedup_bibframe(graph, cache_datastore):
    query = graph.query(SPARQL_PERSON_QUERY)
    for row in query:
        subject = row[0]
        access_point = row[1]
        access_point_digest=hashlib.shal(
            str(access_point).encode()).hexdigest()
```

```

pattern = "*:{:}:{:}".format(
    BF_AUTH_PT_DIGEST,
    access_point_digest)
existing_subjects = cache_datastore.keys(pattern)
if len(existing_subjects) > 0:
    subject_digest = existing_subjects[0].split(":")[0]
    new_subject=rdflib.URIRef(
        cache_datastore.get(subject_digest))
    for pred, obj in graph.predicate_objects(
        subject=subject):
        graph.add((new_subject, pred, obj))
        graph.remove((subject, pred, obj))
    return graph

```

我们的第二个 Python 函数遍历 BIBFRAME 图列表,并运行去重函数 `dedup_bibframe`,然后将每个图的三元组使用关联数据片段服务器的 `add_triple` 函数添加到 Redis 缓存中,最后调用第三个函数 `index_graph` 将图的 JSON 序列化为 ElasticSearch 索引:

```

def process_graphs(graphs):
    for graph in graphs:
        graph = dedup_bibframe(graph)
        for s,p,o in graph:
            add_triple(cache_datastore, str(s), str(p), str(o))
            index_graph(graph)

```

在所有 RDF 图被提取到关联数据片段服务器之后,我们首先为简·奥斯汀和赫尔曼·梅尔维尔计算授权访问点的 SHA1 哈希摘要及 BIBFRAME `authorizedAccessPoint` 的摘要,然后我们就能查询 Redis,看看缓存中是否有重复的 BIBFRAME People:

```

>>> jane_shal_digest = hashlib.shal('Austen, Jane, 1775-1817.'.encode()).
hexdigest()
>>> jane_shal_digest
'4c4da79455d1cee81d7d8737026f0607835f4e77'
>>> herman_shal_digest = hashlib.shal('Melville, Herman,
1819-1891.'.encode()).hexdigest()
>>> herman_shal_digest
'04d0ae092106877146b59ef161409ae25f43df92'
>>> auth_access_pt_digest = hashlib.shal(str(BF.authorizedAccessPoint)).

```



```
encode()).hexdigest()
>>> auth_access_pt_digest
'a548a25005963f85daa1215ad90f7f1a97fbe749'
```

接下来，我们将通过使用每个访问点的 SHA1 查看是否有重复的主语，并构造 Redis 模式获取与之匹配的键。如果我们的去重算法正确执行，那么在使用 Redis 的 KEYS 命令和模式查询 Redis 缓存时，我们理应仅能看到一个三元组：

```
>>> jane_pattern = "*:{:}:{:}".format(auth_access_pt_digest, jane_shal_
digest)
>>> jane_pattern
'*:a548a25005963f85daa1215ad90f7f1a97fbe749:4c4da79455d1cee81d7d8737026f0
607835f4e77'
>>> bibcat_redis.keys(jane_pattern)
[b'2b6f885ab822be23947c5a822b928554cf25d4cd:a548a25005963f85daa1215ad90f7
f1a97fbe749:4c4da79455d1cee81d7d8737026f0607835f4e77']
>>> herman_pattern = "*:{:}:{:}".format(auth_access_pt_digest, herman_shal_
digest)
>>> herman_pattern
'*:a548a25005963f85daa1215ad90f7f1a97fbe749:04d0ae092106877146b59ef161409
ae25f43df92'
>>> bibcat_redis.keys(herman_pattern)
[b'9cf06cced925d745e8bd6ce74ea28950d9a41c64:a548a25005963f85daa1215ad90f7
f1a97fbe749:04d0ae092106877146b59ef161409ae25f43df92']
```

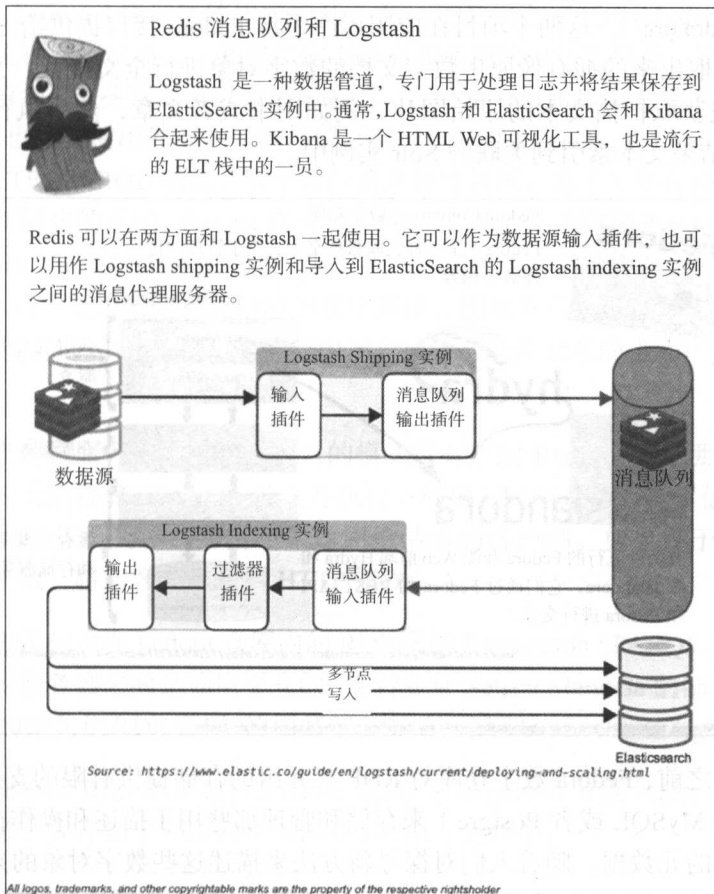
在这个例子中，我们使用 Redis 和关联数据平台片段服务器对 RDF 图进行预处理，再将其序列化到 Elasticsearch 中建立索引。我们会避免为任何暂时的或者不必要重复的信息建立索引，这些信息虽然是缓存的一部分，但是在我们将其中的人去重成为图之前，不应该作为 Elasticsearch 文档建立索引。

ElasticSearch、Logstash 和 Redis

Logstash 是一个开源程序，能从广泛的程序中接收操作日志和错误日志，并将这些日志索引到 Elasticsearch 中，以便更好地对日志进行分析和搜索，同时通过 Kibana 项目提供丰富的可视化界面和报告。ElasticSearch、Logstash 和 Kibana 这三种技术通常被称作“ELK 栈”，是一种非常受欢迎的日志数据可视化组合。Logstash 接收不同的输入数据源，可以通过输入插件进行配置并运行。可以使用 Redis 输入插件接收传入的 Redis 消息，同时 Logstash

会将这些消息索引到 Elasticsearch 中。

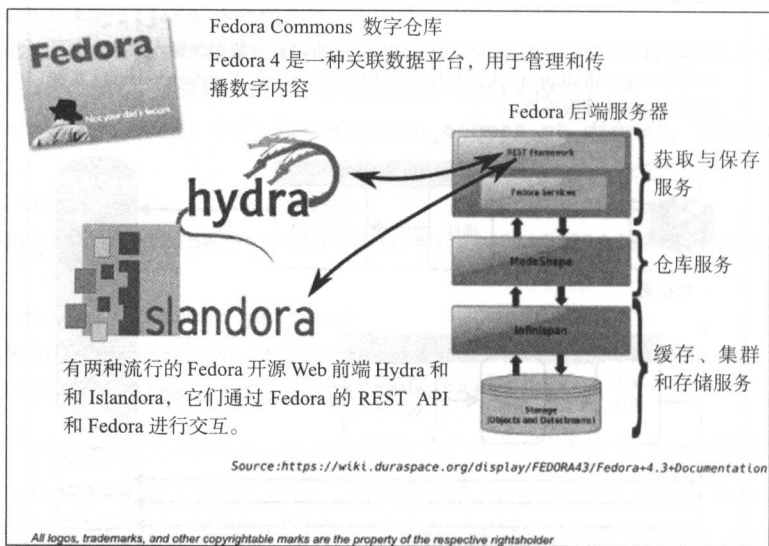
在一些更复杂的 ELK 配置中, Redis 是作为消息队列使用的(我们会在之后的章节中专门详细讲解这一主题)。输入插件会捕捉日志事件, 事件通知会被推送到消息队列中, 然后 Logstash 消息输入插件会接收事件并将其索引至 Elasticsearch 中。Redis 是消息队列的其中一个方案。在配置和部署 Logstash 和 Elasticsearch 时, 配置并激活消息的输出和输入插件是相对简单的。



Redis 作为 Fedora Commons 的智能缓存补充

用于数字图书馆和档案领域的更为专业的 NoSQL 数据存储技术之一是叫作 Fedora

Commons 的开源项目。它是基于 Java 的关联数据平台，用于存储和保存数字对象。通常，我们会将 Fedora Commons 简称为 Fedora（虽然这会引起一定的困扰，因为 Fedora 既指代 Fedora 数字仓库，又指代 Fedora Linux 发行版）。该平台存储的元数据以 RDF 图的方式描述对象。为了充分利用 Fedora Commons 的能力，需要附加的 SPARQL 三元组仓库，以及最流行的 Apache 的 Fuseki 和 Blazegraph。使用 Fedora 作为数字仓库的大多数图书馆和其他文化遗产机构，也会使用下面两种最流行的 Web 展现前端。一个是基于 Druapl 的开源项目叫作 Islandora（<http://islandora.ca>），另一个是 Ruby-on-Rails 开源项目叫作 Hydra（<http://projecthydra.org/>）。这两个项目在应用程序中将 Fedora 接口提供给 Solr 搜索索引，以便针对那些提取出来的拥有像原生数字文档的数字对象进行全文搜索。又或者采用一种工作流程，首先获取扫描文本的原始图片，例如图书或者文章，然后执行光学文字识别（OCR），并将结果文本索引到关联的 Solr 实例中。



在 Fedora 4 之前，Fedora 数字仓库对 RDF 三元组的存储提供有限的支持，需要关系型数据库（通常是 MySQL 或者 Postgre）来存储和管理那些用于描述和操作存储在仓库中的数字对象所必须的元数据。随着人们对探寻新方法来描述这些数字对象的兴趣不断增加，Fedora Commons 社区决定抛开原先的架构，转而直奔一种功能完整的关联数据解决方案。

关联数据起始于一篇文章，请参阅附录中来源的第 7 章中的第五点，由 Tim Berners Lee 先生在 2006 年发表，他列出了 4 种以机器可操作的方式在网络上公开数据的规则。分别是：

- 使用 URI 作为事物的名字
- 使用 HTTP URI 查阅那些名字
- 当某人查阅 URI 时, 使用 RDF 和/或 SPARQL 提供有用的信息
- 将连接包含在其他 URI 中以便人们探索更多的事物

World Wide Web Consortium (W3C) 采用了这些规则并于 2015 年在 <http://www.w3.org/TR/ldp/> 上针对关联数据平台发布了建议。该建议被 Fedora 开发社区作为 Fedora 4 的需求来源。Fedora 4 组件栈使用两种基于 Java 的数据存储技术 ModeShape 和 Infinispan 将对象和数据流存储到磁盘上, 并提供用于访问和展现的 REST 服务。这些 REST 服务允许你创建新的资源。这些资源可以看成是容器, 被描述为与元数据关联的 RDF 图或者二进制文件。Fedora 也需要使用外部三元组存储, 例如最流行的 Apache 的 Fuseki 和 Blazegraph, 两者提供了 HTTP SPARQL 端点, 用于运行请求和更新图。为了方便并减轻保持 Fedora 和三元组存储之间同步的开销, Fedora 将通知事件发布到 Java 消息服务 (JMS) 主题上。JMS 代理提供了 OpenWire 和 STOMP 协议, 用于和许多编程语言进行交互, 例如 Java、Python、Ruby 和 PHP。对更复杂的消息通信应用程序来说, 例如将三元组存储与 Fedora 仓库保持一致, 需要用 Apache Camel 路由来响应用户或者与仓库交互的进程产生的事件的创建、修改或者删除。

在关联数据片段服务器中, 如果传入的模式没有匹配 Redis 缓存, 那么一条 SPARQL 查询将被发往三元组存储 SPARQL 端点并执行。如果三元组存储返回了值, 那么 Redis 缓存将对新的三元组进行更新, 并将结果返回给请求的客户端。如果用于获取匹配信息的 SPARQL 查询失败, 将向客户端返回 HTTP 错误。

为了展示镜像包含在 Fedora 仓库内的 RDF 图的 Blazegraph SPARQL 端点和基于 Redis 的关联数据片段服务器之间是如何交互的, 我们将对 Nelson Mandela 的部分家谱进行建模, 采用 Fedora 中的持久化 RDF 容器, 使用基于 RDF 的词汇 (<http://schema.org/>) 来表示 person。

我们将从简单的 RDF 图开始, 以 Turtle 格式 (采用 person 节点) 表示 Nelson Mandela、它的父亲和母亲, 并存储于 nelson-mandela.ttl 文件中:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix schema: <http://schema.org/> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
```

```

<https://en.wikipedia.org/wiki/Nelson_Mandela> a schema:Person ;
    schema:name "Nelson Rolihlahla Mandela" ;
    schema:parent <http://personuid.info/3eaa6bd7-cfa0-4ded-a8f9-
b745618bb4d2>,
        <http://personuid.info/e8205b68-0af8-4faa-947d-0f22f3a3a77d> .

<http://personuid.info/3eaa6bd7-cfa0-4ded-a8f9-b745618bb4d2> a
schema:Person ;
    schema:name "Nonqaphi Fanny Nosekeni" .
<http://personuid.info/e8205b68-0af8-4faa-947d-0f22f3a3a77d> a
schema:Person ;
    schema:name "Nkosi Mphakanyiswa Gadla Henry" .

```

我们把这个简单的图作为 `mandela_graph` 加载到 Python shell。`mandela_graph` 将 Nelson Mandela 和他的父母联系起来。我们将为每个主语创建子图，并将它们提交到 Fedora Commons 上，Fedora Commons 会自动将 RDF 图添加到 Blazegraph 中：

```

>>> sparql = """SELECT DISTINCT ?subject WHERE { ?subject ?pred ?obj .
}"""
>>> for row in mandela_graph.query(sparql):
    subject = row[0]
    result = requests.post("http://localhost:8080/fedora/rest")
    new_subject = rdflib.URIRef(result.text)
    subject_graph = rdflib.Graph()
    subject_graph.parse(str(new_subject))
    subject_graph.namespace_manager.bind(
        'schema',
        'http://schema.org/')
    subject_graph.add((new_subject, rdflib.OWL.sameAs, subject))
    for pred, obj in mandela_graph.predicate_objects(
        subject=subject):
        subject_graph.add((new_subject, pred, obj))
    update_result = requests.put(str(new_subject),
        data=subject_graph.serialize(format='turtle'),
        headers={"Content-Type": "text/turtle"})

```

在启动关联数据片段服务器实例之后，我们首先从 Blazegraph 中查询家谱应用中所有的三元组，并展示其中一个示例三元组，然后使用我们之前定义好的 `add_triple` 函数从

Python shell 上将每个三元组提取到 Redis 缓存中:

```
>>> result = requests.post(
    "http://localhost:8080/bigdata/sparql",
    data={"query": "SELECT ?s ?p ?o WHERE { ?s ?p ?o .}",
          "format": "json"})
>>> bindings = result.json().get('results').get('bindings')
>>> len(bindings)
122
>>> print(bindings[8])
{'s': {'value': 'http://localhost:8080/fedora/rest/7f/61/e3/d0/7f61e3d0-7e53-4d4f-809f-8158631b1608',
       'type': 'uri'},
 'p': {'value': 'http://fedora.info/definitions/v4/repository#mixinTypes',
       'type': 'uri'},
 'o': {'value': 'schema:Person', 'type': 'literal', 'datatype': 'http://www.w3.org/2001/XMLSchema#string'}}
>>> for row in bindings:
    add_triple(redis_cache,
                row.get('s').get('value'),
                row.get('p').get('value'),
                row.get('o').get('value'))
```

我们将使用 node.js 实现 Web 应用程序, 并使用 N3 (<https://www.npmjs.com/package/n3>) RDF Node.js 库查询关联数据片段服务器。N3 是由 Ruben Verborgh 开发的, 同时也是使用关联数据片段方法访问 RDF 三元组的发起者。首先, 我们将安装该 Node.js 库:

```
$ npm install n3
n3@0.4.3 node_modules/n3
```

为了试验使用该库来解析并添加三元组到关联数据片段服务器上, 我们将运行 Node.js shell 会话, 然后加载 n3 库:

```
$ node
> var N3 = require('n3');
```

现在, 我们来加载核心 Node.js 库 fs, 用来将 nelson-mandela.ttl 文件读取到字符串中:

```
> var fs = require('fs');
> var mandela_ttl = '';

> fs.readFile('nelson-mandela.ttl', 'utf8', function(error, data) {
    if (error) {
        return console.log(error);
    }
    mandela_ttl = data;
});
```

接下来,我们将创建一个 RDF 解析器 n3,解析 mandela_ttl 并将每个 RDF 三元组打印到控制台上(这里只显示前两个三元组,以展示 N3 是如何用 JavaScript 表示三元组的):

```
> var parser = N3.Parser();
> parser.parse(mandela_ttl, function(error, triple, prefixes) {
... if (triple) {
..... console.log(triple);
..... } else {
..... console.log("Finished");
..... }
... });
> { subject: 'https://en.wikipedia.org/wiki/Nelson_Mandela',
  predicate: 'http://www.w3.org/1999/02/22-rdf-syntax-ns#type',
  object: 'http://schema.org/Person',
  graph: '' }
{ subject: 'https://en.wikipedia.org/wiki/Nelson_Mandela',
  predicate: 'http://schema.org/name',
  object: '"Nelson Rolihlahla Mandela"',
  graph: '' }
```

现在,为了准备好示例,我们将回到我们的 Python shell 上,移除 Nelson Mandela 父亲名字的三元组模式:

```
>>> redis_cache.exists(
"56b5bce1875a80f1975edadf3316dc1d0caa1733:30cd0bd17373373839fb3a0ffaa6bba
51a17ba6c:543718498c1fb0ee1fe75744728f22ea25e8d47f")
True
>>>redis_cache.delete("56b5bce1875a80f1975edadf3316dc1d0caa1733:30cd0bd1
```

```
7373373839fb3a0ffaa6bba51a17ba6c:543718498c1fb0ee1fe75744728f22ea25e8d47f")
```

```
1
```

关联数据片段服务器的逻辑处理流程是，如果在初始查询中没有找到结果就查询 SPARQL 数据库。如果在三元组存储中找到三元组，那么该三元组就会被添加到缓存中，并返回给调用方。由于我们移除了父亲名字的 Redis 键，如果我们连接到关联数据片段服务器的 REST API，传入以父亲的 URI 作为主语，并以 `http://schema.org/name` 作为谓语，那么关联数据片段服务器应当查询 Blazegraph SPARQL 端点，将三元组添加回缓存中，并将完整的结果以 JSON 格式返回。

总结

本章一开始研究了数据存储技术，从最流行的支持 SQL 的关系型数据库开始。从关系型数据库开始，我们专门研究了 MongoDB 和 BSON 文档之类的文档型数据存储。接下来是图数据库，之后是全文搜索和键值数据存储，我们特别强调了 Redis。最后我们研究的是宽列存储。

我们总共举了 3 个例子来展示如何将 Redis 用作补充技术。其中一个例子我们使用 MongoDB 来存储使用数据。采用 Redis 分析假想的 MARC21 目录，可以在提升性能的同时降低应用程序的复杂度。第二个例子中，我们将 Redis 用作去重 BIBFRAME RDF 图的预处理器，将关联数据片段服务器当作临时数据存储。第三个例子展示了 Redis 和关联数据片段服务器是如何补充由 Fedora Commons 和 Blazegraph 组合而成的图关联数据平台。

继续沿着 DevOps 的轨迹，第 8 章我们将深入研究 Docker 容器和 Redis 是如何为 IT 运营和开发开启全新的、更好的方式。

8

Docker 容器与云端部署

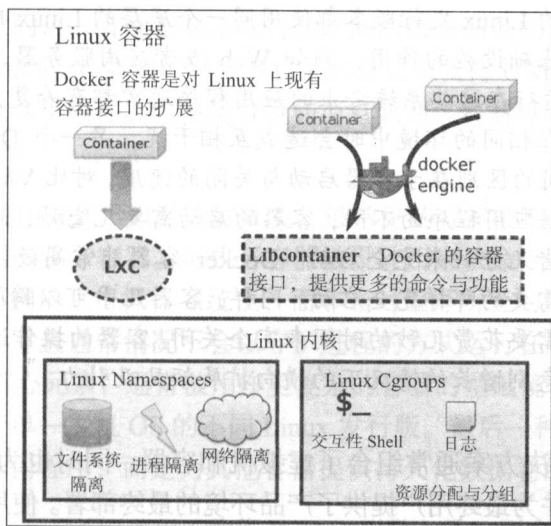
你是否对 Docker 突然流行起来迷惑不解？过去这几年人们对于使用 Linux 容器来构建并运行应用程序的兴趣逐步增长是否也让你不明就里？采用 Docker 会为组织带来真正的效率提升，不仅体现在应用程序的开发，更重要的是应用程序的部署、安全及在生产环境中的恢复。Docker 是基于容器技术的，大致来讲这是一种 Linux “操作系统虚拟化”方法。它允许多个应用程序在单个 Linux 宿主实例上隔离的环境中运行。如果在 Docker 内运行 Redis，那么定义在 Linux 环境中的配置和其他设置选项就能够被复制，并且能独立于其他进程和应用程序。Docker 在提升运维效率方面下了不少功夫，不过将 Redis 数据库托管到公有云或者私有云上倒也是个不错的建议。事实上，大多数流行的云供应商支持在它们的平台上运行 Docker 容器，从而减少组织的工作负载并转至第三方。本章将研究 Redis 在 Amazon、Redis Labs 和 DigitalOcean 这三大供应商上的云托管，并以此作为如何在云端使用 Redis 的示例。

Linux 容器

Docker 容器基于现有 Linux 内核中的功能，例如 cgroups 和 namespaces，这些功能都先于 2013 年 Docker 的第一个版本。在 Linux 内核中，cgroup 可以隔离并限制 CPU、内存、磁盘 I/O，以及网络访问进程，所有这些都受到相同的约束。cgroup 也能从容器中捕捉所有 STDOUT、STDERR 和 STDIN 输出，并存储于容器外部的可访问日志中。与 cgroups 相关，内核命名空间允许进程组进行集群，致使这些进程被隔离开并且无法访问 OS 中的其他资源。具体的 Linux 子系统拥有自己的 namespaces，包括 PID 命名空间、网络命名空间、挂载命名空间、IPC 命名空间和用户命名空间。这些 namespaces 将进程包含到单独的“虚拟”OS 视角，它们甚至无须知道其他系统或者用户进程也运行在同一台 Linux 机器上。虽然有

着像 LXC (<https://linuxcontainers.org/>) 这样的 Linux 容器实现, 但到目前为止最流行的 Linux 容器项目非 Docker 莫属。不过 Docker 也可以将 LXC 当作后台服务使用。Docker 的本地容器称作 libcontainer, 从 Docker 0.9 版本开始使用, 并且支持许多 LXC 不支持的命令。

容器是一种轻量级 Linux 环境, 将应用程序和其所有的依赖封装到一个单独的包中。这个包是可运行的并且能以一致的、可靠的方式进行部署。特别是对那些使用 Redis 的服务器端应用程序及其他 Web 和应用程序服务器技术来说, 容器技术意味着所有这些子系统被打包到一个单一对象中, 该对象能够像单一应用程序那样启动、停止及重启。Docker 公司开发了一套用于支持的开源软件, 现在附加在 Docker Toolbox 中, 其中包括了用于管理容器的 Docker 引擎。Docker 公司也在 <https://hub.docker.com> 上为称作镜像的预构建容器提供了最庞大的源。在该网站上可以找到并下载 Docker 镜像。Docker 爆发性增长的一个原因是围绕容器的平台颇具易用性, 这离不开 Docker 公司为支持此技术而付出的心血。



Docker 容器真正为应用程序提供的是可复制的、可运行于不同上下文的静态运行时环境。运行中的容器共享当前主机 Linux 内核, 而由容器提供自身所需的环境。举例来说, 如果一位程序员使用 Ubuntu 开发一个应用程序, 但是在生产环境中, 应用程序需要运行在 SELinux 环境中, 那么 Docker 允许 Ubuntu 运行在容器内, 而主机运行 SELinux。容器消除了整个复杂的运维问题, 当开发环境和部署环境使用不同的发行版本时, 这些问题就会出现。所有必要的组件连同外部依赖被打包并运行在一个容器中, 之后不用与主机的环境变量或者进程交互, 也不用去修改它们。环境封包消除了在同一运维环境中运行众多应用程

序所带来的一大堆问题和调试噩梦。

Docker 容器默认是写时复制模式 (copy-on-write)。针对容器的变更只作用在本地,不会对主机上的其他容器造成影响。这也意味着由镜像制作的任何新容器不会受到之前从相同 Docker 镜像制作的容器的影响。通过这种方式,我们可以确保从 Docker 镜像制作的每个容器都有可预见和可重复的初始状态。

虚拟机对比容器



第一次听到 Linux 容器时的一种常见的反应是容器和虚拟机到底有何区别?两者之间最基本的区别在于虚拟化发生在软件栈中的哪个位置。虚拟机抽象了操作系统运行时所需的硬件,而容器则为应用程序抽象了操作系统因而运行在栈中更高的级别。容器只提供所需的可执行程序 and 库接口,以便为应用程序模拟操作系统,这得益于所有的 Linux 发行版本都使用同一个底层的 Linux 内核。虚拟机承担着基础设施的作用,例如 Web 或者应用服务器、数据库服务器等。运行在这些系统之上的应用程序通常都有着复杂的交互,它们运行在相同的环境中时会造成互相干扰。另一个 Docker 容器和 VM 之间的区别在于容器启动与关闭的速度,对比 VM 的启动与停止。根据应用程序的不同,容器的启动需要几毫秒,而虚拟机需要几秒或者几分钟来完全启动。Docker 容器非常高效,不需要像虚拟机所需要的那样完全启动。同样,容器几乎可以瞬间关闭,而虚拟机则需要花费几秒的时间来完全关闭。容器的操作速度对于流量和用量急剧增长的情况下的横向扩展颇具吸引力。

大多数 Docker 解决方案通常组合了虚拟机和容器,同时也为应用程序的开发提供了灵活性和实用性,并为最终用户提供了产品环境的最终部署。使用虚拟机的优势在于新机器很容易在几分钟内创建并部署好,对比以前需要花费几个月时间来采购、装机,并在数据中心配置物理服务器。VM 易于创建、移动,并能在运营环境改变时快速移除。同时,VM 能够通过将众多虚拟机运行在单一物理机上来更好地利用硬件,而不用因不同的用途而使用多台服务器。

VM 的增长也推动了计算云(共有的、私有的或者混合的)的增长。单个 VM 镜像可以运行或者部署在多个供应商的云上,例如 EC2、Rackspace、DigitalOcean、Google Cloud 和 Microsoft Azure。VM 也支持不同的计费方式及云计算模型,例如基础设施即服务(IaaS)、

平台即服务 (PaaS) 或者软件即服务 (SaaS)。我们将在本章后面部分介绍几个特定于 Redis 的云选项, 这些选项作用于更高级别的 PaaS 或者 SaaS。

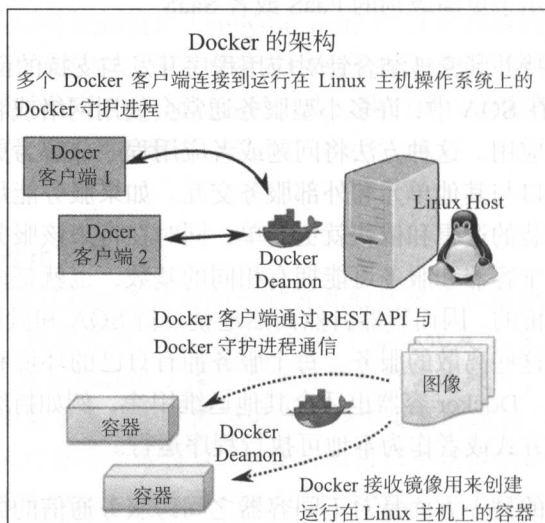
容器也能更好地支持并紧密地结合针对应用程序开发与支持的面向服务的架构 (SOA) 或者微服务设计模式。在 SOA 中, 许多小型服务通常会使用网络或者其他通信协议进行通信, 从而聚集成作为一个应用。这种方法将问题或者应用程序分解为更小的单元。这些单元通过一组良好定义的接口与其他单元和外部服务交互。如果服务能足够独立而显得更为原子化, 那么为该服务封装的逻辑和依赖就更简单, 同时在将来该服务也更有可能被其他组件或应用程序重用。由于容器和服务可能拥有相同的基数, 也就是说服务和运行单一进程的容器在需求方面是等价的。因而容器自然而然地就支持 SOA 和微服务。这种匹配也使得运维团队能更好地支持这些离散的服务。每个服务拥有自己的环境和依赖, 在服务故障时可以进行隔离或者重启。Docker 容器也适合其他运维用途, 例如持续集成 (CI) 平台、伙伴分发, 以云端部署的方式或者作为本地可执行程序运行。

容器技术值得注意的另一方面是对不同容器之间跨服务通信的微调能力。容器间的通信可以发生在本地 IP 回环, 容器间可以共享目录的方式来支持 UNIX 套接字、内存映射文件或者命名管道, 以及共享内存、内核信号量和消息队列。容器可以基于 Linux 命名空间以完全隔离的方式运行。

Docker 容器基于使用方式可以分为好几类。最常见的容器类型是应用程序容器, 可以继续分解为可执行容器和服务容器。可执行容器被设计用来从命令行执行二进制, 同时允许二进制程序运行在不同的主机操作系统上, 而非当时编译所处的原始 OS 上。服务容器封装了应用程序服务, 并且通常情况下会以守护进程的方式运行在后台。机器容器包含的是 Linux 发行版的非核心元素, 通常被用作更复杂的容器的基础镜像, 以及提供一种机制用来运行和测试来自于单一主机 OS 的不同 Linux 发行版。最后一种 Docker 容器类型是数据卷容器, 它不运行任何程序, 而是为其他容器提供持久化数据卷的包装。

Docker 架构是由 Linux 守护进程 (称作 Docker 引擎) 及一到多个通过 REST API 连接到 Docker 引擎的 Docker 客户端组成的。Docker 守护进程监听位于主机 `/var/run/docker.sock` 上的 UNIX 套接字, 并从属于 `docker` 组。Docker 客户端和服务器之间的通信默认没有加密, 因此从授信网络之外开启对守护进程的访问会导致安全危机, 因为外部客户端能以提升的权限来运行进程。对于运行在 Ubuntu 主机上的 Docker 来说, 其守护进程的配置文件存放在 `/etc/default/docker`, 而对于 Red Hat 企业版来说则是存放在 `/usr/lib/systemd/system/docker.service`。对于 Ubuntu 主机来说, Docker 的日志存放在 `/var/log/upstart/docker/log`。此外, Docker 引擎使用目录 `/var/lib/docker` 作为

Docker 的主工作目录:



我们可以使用 `-H` 开关运行客户端，或者设置环境变量 `DOCKER_HOST` 来修改主机和端口的的方式，让我们的 Docker 客户端可以连接到本地或者远程 Docker 守护进程。我们可以通过运行 Bash 命令 `export DOCKER_HOST="tcp://0.0.0.0:4646"` 来更改客户端的默认连接及消息发送的默认端口。通过环境变量 `HTTP_PROXY`、`HTTPS_PROXY` 或 `NO_PROXY` 可以为 Docker 客户端设置代理服务器。

Docker 镜像 是容器模板，可以在本地创建，也可以从本地 Docker 镜像或者企业注册的 Docker 镜像下载，或者从公共的 Docker 镜像资源库里下载。最为活跃且最为庞大的 Docker 镜像资源库是 <http://hub.docker.com/>。Docker 容器是 Docker 镜像的运行实例。在 Docker Hub 上被成千上万的镜像所包含的是那些最流行的 Linux 发行版基础镜像，例如 Ubuntu、Fedora、Debian、CirrOS、CentOS 和 CoreOS，以及那些最流行的应用程序的预装镜像，例如 nginx、WordPress、MongoDB、MySQL 和 Redis。当从镜像启动 Docker 容器时，如果无法在本地找到镜像，那么 Docker 接下来会从 Docker Hub 上寻找匹配的镜像。如果能找到，就下载该镜像并创建容器。

Docker 最初是由 Python 编写的，不过多年来，Docker 公司使用 Go 语言重新实现并改进了 Docker。Go 编程语言是由 Google 开发并支持的。作为基于 Apache 2 License 的开源技术，Docker 可以从 <https://github.com/docker/docker> 上获取。

与 Redis 相关的 Docker 基础

如果你已经是一位 Docker 用户，请跳过本节，因为我们将通过运行官方 Redis Docker 镜像（可从 https://hub.docker.com/_/redis/ 上下载）讲解 Docker 入门的步骤。你可以在 <https://docs.docker.com/installation/> 上找到相应操作系统中 Docker 安装目录的说明。对于 Macintosh 和 Windows 主机操作系统来说，Docker 的安装需要使用专为这些平台设计来运行 Docker 容器的 Docker Toolbox 轻量级 Linux 系统。另外，你也可以使用像 VirtualBox 这样的 VM 管理器来运行 Linux 发行版，再安装和运行 Docker。为了在 Linux 上运行 Docker 后台进程，必须保证 Linux 内核版本较 3.10 版更新且为 64 位。Docker 的最终目标是运行在广大的处理器和操作系统上，包括 Window Server 2016。

在 Linux 上安装 Docker 之后，打开新的终端窗口，将当前用户添加到新的 Docker 组里，这样做可以简化我们的工作，命令如下所示：

```
$ sudo usermod -aG docker {your-username}
```

请确保登出并重新登录，以保证当前用户是新的 Docker 组里的活跃成员。如果是在 Windows 或者 Macintosh 上使用 Docker，那么 Docker Toolbox 会为这些平台设置并加载必要的环境变量。下一步，让我们先看看 Docker 当前的版本，然后启动我们的第一个容器，即 Docker hello-world 镜像：

```
$ docker --version
```

```
Docker version 1.8.2, build 0a8c2e3
```

```
$ docker run hello-world
```

```
Unable to find image 'hello-world:latest' locally
```

```
latest: Pulling from library/hello-world
```

```
535020c3e8ad: Pull complete
```

```
af340544ed62: Pull complete
```

```
library/hello-world:latest: The image you are pulling has been verified.
```

```
Important: image verification is a tech preview feature and should not be  
relied on to provide security.
```

```
Digest: sha256:02fee8c3220ba806531f606525eceb83f4feb654f62b207191b1c92091  
88dedd
```

```
Status: Downloaded newer image for hello-world:latest
```

```
Hello from Docker.
```

```
This message shows that your installation appears to be working  
correctly.
```

如果 Docker 运行正确，用于协调的后台进程会如期工作，并且会运行从网络上下载的容器或者任何新构建的容器。下一步，我们将使用 Docker PULL 命令拉取官方的 Redis Docker 容器：

```
$ docker pull redis
Using default tag: latest
latest: Pulling from library/redis
ba249489d0b6: Pull complete
19de96c112fc: Pull complete
d990a769a35e: Pull complete
.
.
.
library/redis:latest: The image you are pulling has been verified.
Important:
image verification is a tech preview feature and should not be relied on
to provide security.
Digest: sha256:3c3e4a25690f9f82a2a1ec6d4f577dc2c81563c1ccd52efdf4903ccdd2
6cada3
Status: Downloaded newer image for redis:latest
```

我们可以通过使用 Docker ps 命令查看是否有运行着的 Docker 容器：

```
$ docker ps
CONTAINER ID IMAGE COMMAND CREATED
STATUS PORTS
```

目前为止，我们没有启动运行任何 Docker 容器。我们可以通过运行相同的命令但是携带 -a 参数来查看是否有 Docker 引擎管理的现存容器：

```
$ docker ps -a
CONTAINER ID IMAGE COMMAND CREATED
STATUS PORTS NAMES
22c7c12672f hello-world "/hello" 34 minutes ago Exited
(0) 34 minutes ago pensive_pasteur
```

展示在 CONTAINER ID 列下的是 Docker 容器被分配的唯一、缩短版本的 UUID。对于我们的 hello_world 容器来说，缩短了 UUID 为 22c7c12672f。如果你没有为容器指定名字，Docker 引擎会随机创建一个名字，在本例中即 pensive_pasteur。你的

容器很有可能是另一个随机的名字。使用 `--name` 开关允许你为容器显式地设置名字；不过，容器名字必须在主机上运行的 Docker 后台进程范围内是唯一的。我们现在使用 `run` 命令，基于我们之前下载的官方 Redis 镜像启动容器。

我们将传入 `--detach=true` 参数以便在后台运行，使用 `--name=redis` 参数来为我们的容器命名以代替默认的随机名称，并且我们也会传入 `-p 6379:6379` 参数将容器默认的 Redis 6379 端口映射到 Docker 主机的 6379 端口。该命令将返回容器的 `sha1ID`：

```
$ docker run --detach=true --name=redis -p 6379:6379 redis
51fde4c2100f64fbc720fb395e2857be8b98a78e50ba75d0baed89ded4c1b18
```

现在，通过运行 `ps` 命令我们应该能看到运行中的 redis 容器：

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	
51fde4c2100f	redis	"/entrypoint.sh redis"	About a minute ago
	Up About a minute	0.0.0.0:6379->6379/tcp	redis

新打开一个终端窗口，我们将启动 `redis-cli` 实例来看看是否能够连接上运行在 Docker 容器中的 Redis 实例：

```
~/redis/src/redis-cli
127.0.0.1:6379> DBSIZE
(integer) 0
```

我们使用 Docker 的 `stop` 命令来停止 redis Docker 容器，并通过发送 `ps` 命令来确认容器确实没有运行：

```
$ docker stop redis
redid
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS		

Docker 容器也能以交互式模式运行并分配有一个伪 `tty`。这需要向 Docker `run` 命令传递 `-it` 参数，同时取消 `--detach=true` 以便在前台运行容器。现在，在我们继续创建交互模式的新 Redis 容器之前，先将本地环境中的 Redis Docker 容器删除：

```
$ docker rm redis
```


在 bash 上传入路径并运行该命令，你将进入容器的 root 会话。通过该命令提示窗口，我们可以通过展示 /etc/os-release 的文件内容来探索 Redis 容器：

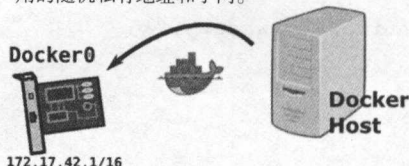
```
root@55ec569c1ded:/data# cat /etc/os-release
PRETTY_NAME="Debian GNU/Linux 7 (wheezy)"
NAME="Debian GNU/Linux"
VERSION_ID="7"
VERSION="7 (wheezy)"
ID=debian
ANSI_COLOR="1;31"
HOME_URL="http://www.debian.org/"
SUPPORT_URL="http://www.debian.org/support/"
BUG_REPORT_URL=http://bugs.debian.org/
```

从这里我们可以看到，我们的 Redis 镜像是基于 Debian Linux 发行版的。从运行着的容器的 root 终端上，我们可以查看容器运行着自己的网络接口：

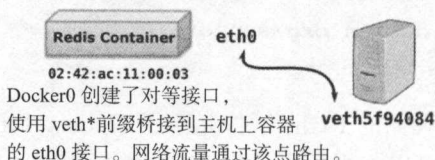
```
root@55ec569c1ded:/data# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
8: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
UP
    link/ether 02:42:ac:11:00:03 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.3/16 scope global eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe11:3/64 scope link
        valid_lft forever preferred_lft forever
```

网络和 Docker 容器

当启动 Docker 时，它会在主机上创建虚拟网络接口 Docker0，并且会选择主机上未使用的随机私有地址和子网。



当容器被创建时，MAC 地址会使用分配给容器的 IP 地址生成，范围从 02:42:ac:11:00:00 到 02:42:ac:11:ff:ff



Docker0 创建了对等接口，使用 veth*前缀桥接到主机上容器的 eth0 接口。网络流量通过该点路由。

最后，我们将检查当前的 /data 目录下的内容，并退出运行中的容器：

```
root@55ec569c1ded:/data# ls
root@55ec569c1ded:/data# exit
```

因为我们没有在容器中启动 Redis，因此 /data 目录是空的，没有包含 dump.rdb 文件。我们现在运行一个新的 Redis 容器，命名为 redis，并以后台的方式运行，同时通过将容器端口映射到主机 6379 端口的方式使其在容器 6379 端口可用：

```
$ docker run --detach=true --name=redis -p 6379:6379 redis
```

在第二个终端窗口中，我们将运行 redis-cli 并添加一个新的键：

```
127.0.0.1:6379> dbsize
(integer) 0
127.0.0.1:6379> set book 1
OK
127.0.0.1:6379> BGSAVE
Background saving started
```

现在，如果我们想连接到运行中的 Redis 容器，可以使用 Docker exec 命令，带上 -i 开关使得可以以交互的方式运行会话，带上 -t 标记可以运行伪 tty 会话以检查 /data 目录下的内容：

```
$ docker exec -it redis /bin/bash
root@e4629ca31026:/data# ls
dump.rdb
```

Docker 命令 `exec` 可以接受另外两个参数，分别为 `-d` (`--detach=true`)，可以将容器以后台的方式运行，以及 `-u` (`--user=`)，在特定用户名或者 UID 下执行命令。

另外一个很有用的用来检查运行时容器的 Docker 命令是 `logs` 命令。`logs` 命令用来展现容器捕获的 `STDOUT/STDERR` 输出。它接收 `--tail` 参数，用来展示日志文件的最后 1 到多行，这和 UNIX 的 `tail` 程序相似。在我们的 Redis 容器上运行该命令，并限制显示日志文件的最后 5 行，得到的输出结果如下所示：

```
$ docker logs --tail 5 redis
1:M 22 Sep 13:44:20.510 * The server is now ready to accept connections
on port 6379
1:M 22 Sep 13:47:22.755 * Background saving started by pid 18
18:C 22 Sep 13:47:22.817 * DB saved on disk
18:C 22 Sep 13:47:22.818 * RDB: 6 MB of memory used by copy-on-write
1:M 22 Sep 13:47:22.849 * Background saving terminated with success
```

另外两个 Docker 命令 `top` 和 `stats` 允许你检测运行中容器的额外运行时状况和环境变量及进程。`top` 命令展示了 Redis 容器中运行的进程：

```
$ docker top redis
UID                PID                PPID                C
STIME              TTY                TIME                CMD
999                15553              696                 0
06:38              ?                  00:00:00            redis-server
*:6379
```

`stats` 展示了 `redis` 容器的实时状态视图。我们可以通过运行 `redis-cli` 会话发送一些命令进行测试，可以看到以下结果：

```
CONTAINER          CPU %              MEM USAGE/LIMIT    MEM %
NET I/O
redis              0.20%              6.619 MB/1.579 GB  0.42%
1.296 kB/5.044 kB
```

`start`、`restart` 和 `attach` 及 `stop` 这些 Docker 命令可以让你更好地管理可能驻留

在本地 Docker 资源库里的 Docker 容器。首先，我们将停止运行 Redis 容器：

```
$ docker stop redis
```

```
Redis
```

然后，我们将启动 Redis 容器，并通过 ps 命令确定 Redis 仍然运行着：

```
$ docker start redis
```

```
redis
```

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	
e4629ca31026	redis	"/entrypoint.sh redis"	46 hours ago
Up 3 seconds	0.0.0.0:6379->6379/tcp	redis	

使用 Docker 命令 attach 将在前台运行容器，同时展示 STDOUT/STDERR 输出。注意，如果你向当前活动窗口发送了 `Ctrl+C`，就会向容器发送 kill 信号 (SIGINT)，使其停止运行：

```
$ docker attach redis
```

```
^C1:signal-handler (1443097862) Received SIGINT scheduling shutdown...
```

```
1:M 24 Sep 12:31:02.861 # User requested shutdown...
```

```
1:M 24 Sep 12:31:02.861 * Saving the final RDB snapshot before exiting.
```

```
1:M 24 Sep 12:31:02.864 * DB saved on disk
```

```
1:M 24 Sep 12:31:02.864 # Redis is now ready to exit, bye bye...
```

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	

为了避免在附加到容器之后关掉它，你可以使用 `Ctrl + P` 和 `Ctrl + Q` 的键组合从运行中的容器分离。如果在容器初始运行时将 `--sig-proxy` 标识设置为 `false`，`Ctrl + C` 才有用。在 Redis 容器停止之后，我们可以发送 restart 命令激活并运行 Redis 容器：

```
$ docker restart redis
```

```
redis
```

```
$ docker ps
```

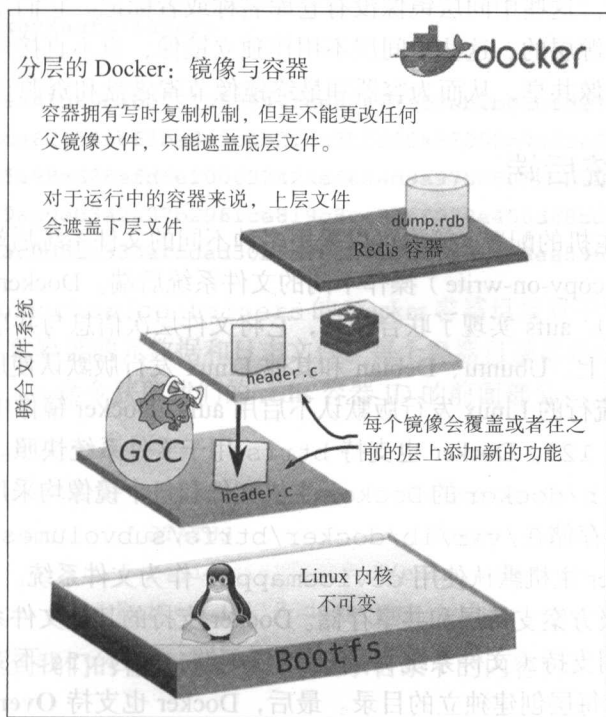
CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	
e4629ca31026	redis	"/entrypoint.sh redis"	46 hours ago
Up 4 seconds	0.0.0.0:6379->6379/tcp	redis	

在了解了这些核心命令来管理 Docker 容器之后，我们将移步到下一个重要的 Docker 组件，理解并创建镜像来为容器提供运行时模板。

Docker 镜像中的层

虽然将应用程序放到 Docker 容器中运行非常棒，但是只有和 Docker 镜像（容器创作的模板）组合起来才能逐步显现 Docker 的优势。Docker 镜像是通过向先前存在的文件系统层中添加新的文件系统层的方式构造的。每一层都有其静态特征，例如应用程序或工具的可执行文件、库及其他配置文件。上层文件路径如果匹配了下层先前存在的文件，就会从执行代码中遮盖掉（mask）这个文件。

举例来说，假设启动了现有的官方 Redis 层，添加自有 `redis.conf` 配置文件的新层会遮盖存放在相同文件路径位置上之前已经存在的 `redis.conf` 文件。



Docker 镜像和容器的分层

一个 Docker 基础镜像没有父亲，通常是由操作系统（如 Ubuntu 或者 CentOS）及根文

件系统组成。为了安全起见，你可以创建一个空的 Docker 基础镜像，然后往里面添加运行应用程序所需的文件，作为彼此独立且互不相同的层。镜像中的文件系统是只读的，运行中的容器无法修改低层镜像的文件，只能修改并保存，然后遮盖低层镜像的文件。镜像文件系统中的不可变文件使得镜像能够提供一致和可重复的环境来运行多个容器并得到一致的结果，同时也减少了容器的磁盘和内存占用，因为它们共享了 Docker 主机相同的父镜像。

当容器第一次启动时，它的文件系统初始是空的。在新运行的容器中任何运行进程的写操作都会保存在容器的文件系统中。就像之前提到的那样，任何匹配低层镜像层的现存文件都会被遮盖。容器文件系统只包含当前文件和容器镜像层中任意底层文件系统状态的变更。从上至下，容器做出的所有更改及所有现有的镜像层文件系统共同成为联合文件系统（union filesystem）。栈中的最底层称作 bootfs，并将内存文件系统接口供给 Linux 内核，同时 bootfs 也将内核库接口供给上层镜像进程。

除了 Docker 的基础镜像外，每个 Docker 容器和镜像都有一个父镜像。Docker 镜像由中间层镜像构建而成。这些中间层镜像没有仓库名称或者标记。它们是一起用来为容器或者镜像的父镜像提供源层的。这些中间层不用作独立镜像，也不直接作为容器使用，但是可以被派生的后代镜像共享，从而为容器和最终镜像节省磁盘和资源空间。

Docker 文件系统后端

取决于 Docker 主机的配置，镜像可以采用多种不同的文件系统后端，而运行时的容器则会执行写时复制（copy-on-write）操作不同的文件系统后端。Docker 推荐使用高级多层联合文件系统（aufs）。aufs 实现了联合挂载，它将文件层次信息与共享存储及其他文件系统存储在单个挂载点上。Ubuntu、Debian 和其他 Linux 发行版默认使用 aufs。Red Hat 和 CentOS 这两个非常流行的 Linux 发行版默认不启用 aufs。Docker 镜像中的层级数量受限于 aufs 默认的层级数值 127。Docker 也支持 btrfs 用于文件系统快照。块级方案及共享存储需要位于 `/var/lib/docker` 的 Docker 主机文件和每个镜像均采用 btrfs 文件系统，同时容器层作为子卷存储在 `/var/lib/docker/btrfs/subvolumes`。运行着 Red Hat 和 CentOS 系统的 Docker 主机默认使用 devicemapper 作为文件系统。devicemapper 像 btrfs 那样采用块级方案支持层和共享存储。Docker 支持的其他文件系统还有 vfs，用于 Docker 主机上的通用支持（文件系统）。不过这还不够，因为 vfs 不支持层级快照，而是使用父层的深拷贝为每层创建独立的目录。最后，Docker 也支持 **OverlayFS**，用来联合挂载其他文件系统。

`docker images` 命令展示了当前 Docker 主机上所有的镜像。以下是示例环境中的运

行结果展示：

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID
CREATED	VIRTUAL SIZE	
<none>	<none>	
1bbc3672404f	2 weeks ago	521.9 MB
java 8-jre		
81f1a5272622	2 weeks ago	487.9 MB
redis latest		
2f2578ff984f	2 weeks ago	109.2 MB

Docker 镜像存储在 Linux 主机的默认路径 `/var/lib/docker/aufs/layers` 上。在我们的示例环境中，切换到 `root`，更改目录，然后显示以下内容：

```
$ sudo su
```

```
# cd /var/lib/docker/aufs/layers
```

```
# ls
```

```
00db3659acd05f0a98a41d69cab0791055844fcee84f7f53ab2b0cbfd27cb9ae
017d6be562b544d03de624546b63ba8e9c0b21ce3bfd05a32058e9b39efc8672
0225617d4328e423e5e98ad28efd6e10063242aafaeaad9a9758865f026b0a732
038233a03eefb40279ac0eb3a2a87b2961ce819c8ca9c6f938e456d68bde6297
04ac98492065dc05dac0d5da333afcdad50b4e886b9efc3599ea48ea39683ea0
```

位于 `/var/lib/docker/containers` 的 Docker 容器目录对于每个容器来说都有一个独立的目录，保存着容器的元数据和日志文件。每个容器目录都是唯一 ID。我们可以通过运行 `docker ps -a` 命令获取我们的 redis 容器 ID 的前面部分：

```
# docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
e4629ca31026	redis	"/entrypoint.sh redis"	5 days ago	Exited (0) 3 days ago		redis

现在，我们切换到我们的容器目录中，并展示目录下的内容：

```
# cd /var/lib/docker/containers/
```

```
e4629ca310264f7f4a930dcda5f5f8a91710b8a2fe5109996dffdf9adbfd5c5a8/
```

```
# ls
```



```

config.json
e4629ca310264f7f4a930dcdaf5f8a91710b8a2fe5109996dffdf9adbfd5c5a8-json.log
hostconfig.json
hostname
hosts
resolv.conf
resolv.conf.hash

```

config.json 文件包含了 Redis 容器的运行状态和环境变量。json 日志文件存储了容器的活动日志，它和运行 Docker 命令 `docker logs redis` 展现的内容一致。

Docker 中运行态容器生成增量的写时复制 (COW) 数据，存储在 Docker 主机上。这些数据的来源要么是修改了父层中存在的文件，要么是容器中的进程创建的新文件。这些差异存储在 `/var/lib/docker/aufs/diff/` 目录中。为了查看这些变更，我们使用 `docker diff` 命令展示容器文件系统层中的差异。Add (A)、Delete (D) 和 Change (C) 是三种类型的增量，在容器运行时被捕获和展现。首先，我们在新的 redis 容器上运行 diff 命令，会发现没有东西修改过：

```

$ docker diff redis
$

```

下一步，我们打开 `redis-cli` 会话，添加一个键，并发送一个 `BGSAVE` 命令持久化 RDB 快照到容器的文件系统层：

```

$ redis/src/redis-cli
127.0.0.1:6379> SET person:1 "Lucy van Pelt"
127.0.0.1:6379> GET person:1
"Lucy van Pelt"
127.0.0.1:6379> BGSAVE
Background saving started

```

有两种方式来创建 Docker 镜像：一种是使用修改过的运行时容器的快照；另一种是从 Dockerfile 构造一个全新的 Docker 镜像。Dockerfile 是一段包含了 Dockerfile 具体命令列表的简单文本格式。第一种制作镜像的方法，即获取运行时容器的快照是最简单的方法。举例来说，如果我们想要创建基于官方 Redis 镜像自定义镜像的话，只需要加载一些数据，然后保存为新的镜像。我们将介绍这几个步骤。

首先，我们停止并移除 redis 容器：

```
$ docker stop redis
```

```
redis
```

```
$ docker rm redis
```

```
Redis
```

然后，我们以后台运行的方式启动新的 Redis 容器实例，然后通过 `docker exec` 命令连接到容器：

```
$ docker run --detach=true -p 6379:6379 --name=redis redis
2f0b562fe09c4b9663cf3e122d3256ecaf96773c459536dcb9674fd0d347ce26
$ docker exec -it redis /bin/bash
root@2f0b562fe09c:/data#
```

运行中的 Docker 容器没有包含任何数据，因此我们将再打开一个命令行，启动 `redis-cli` 实例，确保 Redis 数据库为空，然后加载一些数据到自定义 Redis 镜像中：

```
$ redis/src/redis-cli
127.0.0.1:6379> DBSIZE
(integer) 0
127.0.0.1:6379> MSET ichi one ni two san three shi four go five roku six
shichi seven hachi eight kyuu nine ju ten
OK
```

在保存了 10 个键（这些键是日语中从 1 到 10 的计数，同时将英文的翻译作为字符串值）之后，我们回到第一个终端窗口，确认现在在 `data` 目录下存在 `dump.rdb` 文件：

```
root@2f0b562fe09c:/data# ls
dump.rdb
```

另一个用来创建镜像，包含持久化数据文件 `dump.rdb` 的方法并不太理想。首先我们退出正在运行的 `redis` 容器，然后发送 `Docker commit` 命令，传入作者和消息参数，创建自定义 Redis 镜像到 `redis-japanese-numbers` 仓库，然后通过查看镜像列表确认我们的新镜像确实创建成功了：

```
root@2f0b562fe09c:/data# exit
exit
$ docker commit --author="Jeremy Nelson" --message="Japanese Numbers"
redis jermnelson/redis-japanese-numbers
6ef38a2d2efb5253db128d4fbaad6379679c2dc5d533e0b54750e1653e038cae
```

```
$ docker images
```

REPOSITORY		TAG	IMAGE ID	
CREATED	VIRTUAL SIZE			
jermnelson/redis-japanese-numbers	latest	6ef38a2d2efb	2	
minutes ago	109.2 MB			
redis	latest	2f2578ff984f	3	
weeks ago	109.2 MB			

现在，我们将停止 redis 容器，并基于新的 redis-japanese-numbers 镜像启动新的 Docker 容器：

```
$ docker run --detach=true -p 6379:6379 jermnelson/redis-japanese-numbers
6537f3e95269439aa976d8220a4c7f6b7c1815ba19fd04814d7c9415f3ae6571
```

为了检测新的镜像中是否包含了之前保存的数据，我们将通过 redis-cli 来检测：

```
$ redis/src/redis-cli
127.0.0.1:6379> dbsize
(integer) 0
```

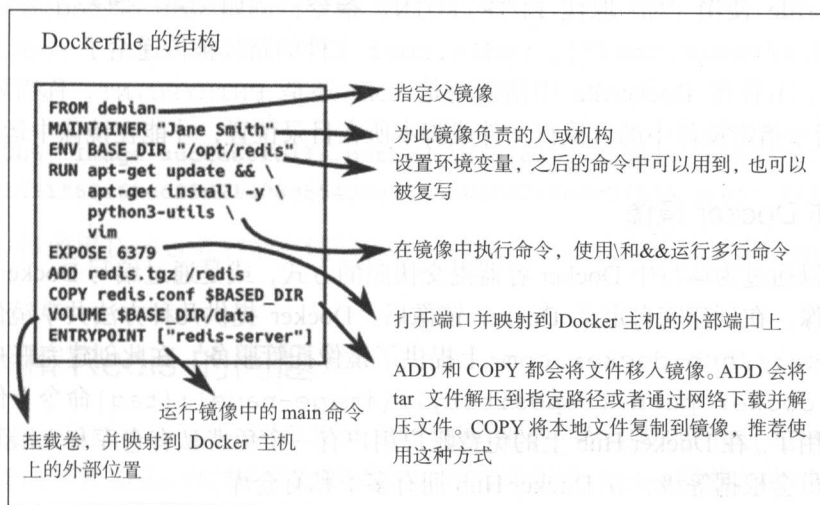
怎么回事，我们的数据呢？这是常见的“疑难杂症”，因为基础 redis 镜像将 /data 目录下的数据挂载到数据卷上。当启动容器时，容器中任何卷都不会被保存下来。这是由 Docker 通过卷语法对数据的持久化与分片的方式决定的。将一个运行时容器提交为新的镜像将不会持久化任何挂载的卷。

这是 Docker 的一个特性。它意味着如果我们想要持久化 rdb 文件，我们就两个选择，要么以区别于 /data 的目录来保存 dump.rdb 文件的方式重启 Redis，要么使用 Dockerfile 创建 Docker 镜像。

使用 Dockerfile 构建镜像

就如我们在之前的章节中看到的那样，从运行中的容器镜像来创建 Docker 镜像是一个手工活，需要连接、安装、然后提交更改才能获得一个可工作的镜像。幸运的是，这个更受欢迎的创建 Docker 镜像的方法是使用基于文本的 Dockerfile 来创建镜像的，以替代在运行中容器上提交更改的方式。Dockerfile 是由一系列指令组成的。镜像的创建依据的是这些命令在文件中的顺序。每个命令在 Docker 镜像中创建一个独立的层，因而需要花点心思来最小化命令的总数。太多的层会导致臃肿的镜像，会花费较长的时间构建、下载，或者上传到 Docker 主机上。Dockerfiles 中第一个命令是 FROM 命令，它指定了用于构建新镜像所

基于的父镜像,大多数情况下是操作系统镜像。当前推荐的操作系统基础镜像是使用 Alpine Linux 镜像,它提供了完整的 OS,并且针对于 Docker 基础镜像做了优化,不到 100M 的大小:



在 Dockerfile 中, RUN、ADD 和 COPY 命令完成了大部分的工作。RUN 指令在镜像中执行一到多个 Linux 命令,对在应用程序中安装包和其他依赖十分有用。多条命令可以通过将两个与符号(&&)拼接起来形成单个 RUN 指令。还有个建议是将这些命令使用反斜杠(\)字符放在不同的行上。ADD 和 COPY 命令都会将文件移入镜像。但是,ADD 提供了额外的功能,TAR 文件会自动解压到本地本件系统,此外还能从 URL 下载文件。COPY 命令将本地文件复制到主机上相对的 Dockerfile 路径中,并将这些文件保存至镜像中。如果想排除有些文件在 COPY 命令之外,可以在 Dockerfile 所在的根目录创建 .dockerignore,这和 .gitignore 文件类似,COPY 命令不会将匹配 .dockerignore 内的模式的文件复制到镜像中。

VOLUME 和 EXPOSE 命令为 Docker 主机和镜像及由镜像制作的一系列容器提供了外部访问。VOLUME 命令为镜像文件系统和 Docker 主机之间提供了一个挂载点,同时作为主机上的一个目录或是文件,能在之后用于保存和持久化容器会话之间的数据。EXPOSE 指令指定了镜像的内部端口,当容器启动时可以通过 -p 参数映射到 Docker 主机端口上。这种用法可以从我们之前的示例中看到,当时我们以 6379 端口在主机上运行的 redis 镜像映射到镜像中相同的端口。

ENTRYPOINT 和 CMD 指令是相关的,但是对镜像提供了不同的功能。当通过镜像运行

容器时，`ENTRYPOINT` 指令指定了默认的可执行文件。也可以在 `ENTRYPOINT` 中使用脚本文件。`bash` 脚本中包含了配置和设置指令，并在之后调用主可执行文件启动。另外，`CMD` 指令允许在镜像内指明要运行的软件，并可向软件的可执行文件传递参数。官方的 Redis Dockerfile 使用 `CMD` 取代 `ENTRYPOINT` 指令，例如 `CMD ["redis-server", "/etc/redis/redis.conf"]`，`redis.conf` 文件的路径被传递给了 `redis-server` 可执行文件。不管在 Dockerfile 中使用的是 `CMD` 还是 `ENTRYPOINT`，你需要通过使用 `WORKDIR` 指令指定镜像中的可执行文件或脚本所在目录位置，才能在镜像中运行命令。

托管并发布 Docker 镜像

我们可以通过为运行中 Docker 容器提交快照的方式，或是通过编写 Dockerfile 的方式构建新的镜像。在创建完自定义 Docker 镜像后，Docker 提供几种方法共享镜像。Docker 公司在 <https://hub.docker.com/> 上提供了镜像托管服务。在此创建完账户后，你可以通过使用 `docker push {repository}/{image-name}:{tag}` 命令上传镜像，之后就可以使用了。在 Docker Hub 上的免费账户用户有一个免费私有仓库和无限制的公共镜像。付费会员会根据等级，在 Docker Hub 拥有多个私有仓库。

如果你不愿意使用 Docker Hub，但又想为组织中的其他成员提供镜像，一部分 Docker 公司在 <https://github.com/docker/distribution> 上的附属项目 Docker Toolkit 允许你托管自有仓库。该项目采用的机制是 Docker 镜像的方式一点都不让人意外，在 Docker 主机上运行下列命令就可以启动：

```
$ docker run --detach=true -p 5000:5000 --restart=always --name registry
registry:2
```

从该 Docker 命令中可以看到，我们以后台的方式运行 `registry` 镜像的容器实例，并且端口为 5000。新的参数 `--restart` 设置成了 `always`，因此如果该容器故障了，那么 Docker 主机会自动重启它。由于注册在了 5000 端口，如果你想将自定义 Redis 镜像推到本地容器，你需要为之打上指向该 `registry` 的标签，在本例中即为 `localhost:5000`。如果你想要在组织中将私有 `registry` 分享给他人，你会需要一台专门的服务器，并设置好网络名称。首先通过为官方 Redis 镜像打上标签将其推送至 `registry`：

```
$ docker tag redis localhost:5000/redis
```

在为 `redis` 镜像打上标签后，我们可以从 5000 端口将镜像推送至本地运行的仓库：

```
$ docker push localhost:5000/redis
```

```
The push refers to a repository [localhost:5000/redis] (len: 1)
2f2578ff984f: Image successfully pushed
54647d88bc19: Image already exists
ed09b32b8ab1: Image already exists
.
.
.
ba249489d0b6: Image successfully pushed \nlatest: digest: sha256:1b47e11f
b5d6395aa1631f60e61cc92d21308d55485e1316c8c8421fc4c07385 size: 34407
```

registry 镜像用 Docker 卷容器存储所有有关本地 Docker 仓库托管镜像的 registry 信息。

Docker 和 Redis 的问题

在讲 Redis 集群和 Sentinel 的章节中，我们没有提到一些主要的问题，这些问题会在使用 Docker 尝试部署 Redis 的高可用方案时发生。当启动新的 Docker 容器并使用 `-p` 指令时，Docker 会执行动态端口重新分配。Sentinel 的自动发现其他运行 Sentinel 的过程，以及从主节点发现从节点列表都假设端口是固定的。如果 Docker 容器内部运行的 Sentinel 映射到了不同的端口，该功能就失效了。

为了在 Docker 中使用 Sentinel，你有两个选择：第一种是为每个运行 Sentinel 的 Docker 容器更新 `sentinel announce-ip` 和 `sentinel announce-port`，以便 Docker Sentinel 向其他运行中的 Sentinel 实例广播（或者说是宣告）正确的 IP 地址和端口号。第二种（如果你是从头开始设置 Sentinel 和配置 Redis，这可能是最为简单的实现方式了）就是要么使用 `-p` 参数（也就是说，当）将主机和容器的相同端口做映射，要么传入 `--net=host` 参数来作自动映射。使用第二种方法可能会有些限制，你将无法在同一个端口上运行多个 Docker 容器，其中每个容器上都运行着 Sentinel，这是因为每次只有一个容器能被映射到 Docker 主机端口上。

使用 Docker Compose 打包应用程序

将应用程序分解来高效使用 Docker 通常需要运行多个容器，每个容器运行着不同的 Docker 镜像。如果你的应用程序拥有多个容器并且互相连接，手工协调应用程序的 Docker 容器管理（启动、停止等）会非常耗时并且容易出错，特别是你会忘记特定容器的必要参

数（假设你有一个卷容器连接到应用程序容器，而应用程序容器需要不同的端口映射）。幸运的是，Docker 公司赞助的开源项目 Docker Compose 可以缓解许多这类问题。

如果你安装了 Docker Toolbox 的话，就自动包含了 Docker Compose。你也可以依照 <http://docs.docker.com/compose/> 上的说明进行安装。为了演示 Docker Compose 的使用，我们将创建一个非常简单的 Flask Web 应用程序，仅仅用于以 HTML 文档的方式展现 Redis INFO 命令的输出。同时，我们也会演示 Docker 很棒的容器间通信的功能。我们使用 Docker Compose 将应用程序容器连接到之前创建的 Redis 容器上。

让我们开始吧。以下是我们的 Flask 应用程序的 Python 源代码，名字叫作 `info.py`。在引入了 Flask、`render_template` 函数和 Python Redis 客户端之后，创建了应用程序和 Redis 客户端实例。请注意，Redis 客户端的主机名为 `redis`，也是我们在 Docker Compose YAML 配置文件中将要连接的 Redis 容器的名字。该应用程序有单一函数 `default`，它将返回 HTML 模板（而非显示），将执行 `info` 命令的结果以格式化表格的形式展示给请求的 Web 浏览器：

```
from flask import Flask, render_template
import redis

app = Flask(__name__)

redis_db = redis.StrictRedis(host='redis')

@app.route("/")
def default():
    return render_template(
        'index.html',
        info=redis_db.info())
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5001, debug=True)
```

在（`info` 应用程序）同一个根目录下，我们创建最小化的 Dockerfile，扩展自 Python 3.4 基础镜像，暴露 5001 端口，并在最后使用 Dockerfile CMD 命令执行我们的代码文件：

```
FROM python:3.4.3
RUN pip3 install flask redis
COPY . /info_app
WORKDIR /info_app
```

EXPOSE 5001

CMD ["python", "info.py"]

创建 Docker Compose 项目的下一阶段是创建名为 `docker-compose.yml` 的 YAML 配置文件。配置文件中的第一部分将为应用程序定义新的容器，我们称之为 `info`。在 `info` 这一部分，`build` 指令引用了我们创建的 `Dockerfile`，同时 `links` 指令列出了 Redis 容器的名称。在 `ports` 指令中，我们将应用程序运行时的 5000 内部端口映射到 Docker 主机的 8080 端口。最后，我们基于 Docker Hub 上的官方 Redis 镜像定义我们的 Redis 容器。`docker-compose.yml` 文件内容如下所示：

```
info:
  build: .
  links:
    - redis
  ports:
    - "8080:5001"
redis:
  image: redis
```

现在必要的准备工作都已经完成，我们可以尝试使用 `docker-compose` 构建应用程序：

```
$ docker-compose build .
redis uses an image, skipping
Building info...
Step 0 : FROM python:3.4.3
---> 575cb3ad9b67
Step 1 : RUN pip3 install flask redis
---> Running in 5342e1c49874
Collecting flask
  Downloading Flask-0.10.1.tar.gz (544kB)
Collecting redis
  Downloading redis-2.10.3.tar.gz (86kB)
Collecting Werkzeug>=0.7 (from flask)
  Downloading Werkzeug-0.10.4-py2.py3-none-any.whl (293kB)
Collecting Jinja2>=2.4 (from flask)
  Downloading Jinja2-2.8-py2.py3-none-any.whl (263kB)
Collecting itsdangerous>=0.21 (from flask)
```

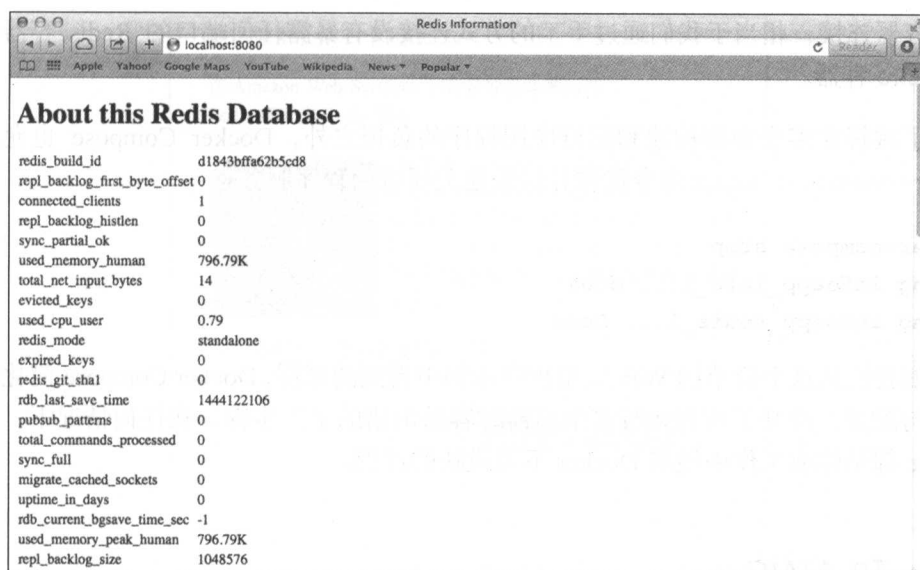


```
Downloading itsdangerous-0.24.tar.gz (46kB)
Collecting MarkupSafe (from Jinja2>=2.4->flask)
Downloading MarkupSafe-0.23.tar.gz
Installing collected packages: Werkzeug, MarkupSafe, Jinja2,
itsdangerous, flask, redis
Running setup.py install for MarkupSafe
Running setup.py install for itsdangerous
Running setup.py install for flask
Running setup.py install for redis
Successfully installed Jinja2-2.8 MarkupSafe-0.23 Werkzeug-0.10.4
flask-0.10.1 itsdangerous-0.24 redis-2.10.3
--> 21d5378d91b3
Removing intermediate container 5342e1c49874
Step 2 : COPY . /info_app
--> b65c476ad781
Removing intermediate container 001debb1ff52
Step 3 : WORKDIR /info_app
--> Running in 59a5a7f29c4a
--> 5a7557640f84
Removing intermediate container 59a5a7f29c4a
Step 4 : EXPOSE 5001
--> Running in 8ccd0b16a8a7
--> 9451da0dc4ba
Removing intermediate container 8ccd0b16a8a7
Step 5 : CMD python info.py
--> Running in 59e66bc7787c
--> 8e165172aa53\nRemoving intermediate container 59e66bc7787c
Successfully built 8e165172aa53
```

这样就启动并构建了应用程序的镜像，之后我们就能够在命令行上使用 `docker-compose up` 命令，并传入 `-d` 参数，以后台运作的方式启动应用程序：

```
$ docker-compose up -d
Starting infoapp_redis_1...
Starting infoapp_info_1...
```

现在，我们可以通过打开 Web 浏览器并访问 `http://localhost:8080/`，看看我们的应用程序是否正常显示，展现的内容如下截屏所示：



我们在 Docker 主机上检查发现有两个容器正在运行中：

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED
b4ac9d5a18dc	infoapp_info	"python info.py"	7
minutes ago	Up 2 minutes	0.0.0.0:8080->5001/tcp	infoapp_
info_1			
6662e75f3ffc	redis	"/entrypoint.sh redis"	37
minutes ago	Up 2 minutes	6379/tcp	infoapp_
redis_1			

我们发现两个容器在后台运行，Docker-compose 顺便为这两个容器分别命名为 infoapp_info_1 和 infoapp_redis_1。

现在，我们将开启另一个终端窗口，使用 redis-cli 连接到 Redis 数据库：

```
$ redis/src/redis-cli
127.0.0.1:6379> INFO
Error: Server closed the connection
```

发生了什么？Docker 的一个出色的功能就是能将容器中的服务与外界进程隔离，但仍然可以通过连到我们的应用程序容器的方式进行连接并使用服务。Docker Compose 处理所

有底层容器连接，相当于我们通过手工的方式连接没有暴露任何端口的 Redis 容器的方式来运行 info 容器。

除了减轻在多个容器构建和运行应用程序的负担之外，Docker Compose 也允许使用 `docker-compose stop` 命令优雅并轻松地关闭应用程序服务：

```
$ docker-compose stop
Stopping infoapp_info_1... done
Stopping infoapp_redis_1... Done
```

就如我们从这个简单的 Web 应用程序示例中看到的那样，Docker Compose 简化了容器的管理与配置，避开了所有运行多个容器时容易出错的手工步骤。在任何情况下，Docker Compose 都是你在工作中使用 Docker 不可或缺的伴侣。

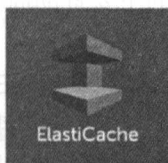
Redis 和 AWS

Amazon Web Services (AWS) 通过它的 ElastiCache 服务（详情请参考 <https://aws.amazon.com/elasticache/>）提供了基于 Redis 协议的内存缓存。你可以使用 Redis 命令，不过背后使用的是 Amazon 自己的后端服务。该服务还包含了自动化常见虚拟机任务的功能，例如补丁管理、故障检测与恢复。相较于其他选择，使用 ElastiCache 更易于扩展，因为你可以通过 AWS 管理控制台配置并添加更多的缓存节点来应对顾客或者组织内部客户不断增加的需求所导致的应用程序运维需求。ElastiCache 的定价会根据你的使用而不同，你可以仅为应用程序运营期间所消费的 ElastiCache 节点资源付费。由于 ElastiCache 只是兼容 Redis 协议，后端并没有真正使用 Redis 服务器，因此 Redis 客户端在使用 ElastiCache 时有几点限制需要考虑。

首先，当前 ElastiCache 限定于 Redis 2.8 及更早的命令与功能，因此所有 Redis 集群的命令将不能使用。如果你的应用程序需要通过多个 Redis 主节点进行分片，那么使用 ElastiCache 意味着你需要为数据使用完全基于客户端的分片方法，而非 Redis 集群。ElastiCache 提供的节点类型与 Redis 的主从实例类似。

在 AWS 上运行 Redis

在 Amazon Web Services 上有多种选择来运行 Redis 或者类 Redis 服务。



ElastiCache 是 Amazon 自有的缓存服务，它提供了 Redis 2.8 协议，对应用程序来说就像是真实的 Redis 实例一样。但是该服务的背后并非真正的 Redis，你无法为应用程序调整配置，也不能更新到最新版本的 Redis。

另一个选择是启动 EC2 虚拟机，并进行手工安装、配置并管理 Redis。



All copyright and logos are property of Amazon

在 AWS 上运行 Redis 的另一个选择，同时也是很多人都会采用的是，在 Amazon 的弹性计算云（EC2）上选购一台硬件虚拟机（HVM）。在 EC2 虚拟机上运行 Redis，你想要最小化 Redis 调用过程来存储到磁盘的延迟，需要选择多核的 EC2 实例，这样每个 CPU 核心的负载就能最小化。你需要禁用 OS 交换机制，那是因为如果 Redis 实例超过了可用 RAM，如果开启了 OS 交换机制，那么你的 Redis 实例就会变得非常慢。这是因为 Redis 通过交换的方式访问和写数据到磁盘。许多组织和个人将他们的开发和运维的 Redis 数据库运行在 AWS 上专门的 EC2 VM。Amazon 激进的定价策略使得在 AWS 上使用 Redis 非常有吸引力且令人信服。

专门的云托管选项

有许多公司提供专门的 Redis 托管，你无须再担心运行 Redis 的操作系统和环境的操作细节。取而代之的是可以快速启动并使用 Redis，无须操心虚拟机或者其他系统管理的任务。在这里就不对所有的 Redis 服务供应商做介绍了，我们将选取其中最为流行的两家专门的 Redis 托管供应商，一家来自 Redis Labs（详情请参考 <https://redislabs.com/>），另一家是来自 DigitalOcean 的 Redis 托管服务。

Redis Labs

2015 年,既是 Redis 的作者又是主要开发者的 Salvatore Sanfilippo 接受了 Redis Labs 的职位引领开源发展,同时 Redis Labs 也成为了 Redis 的主要赞助商。Redis Labs 的总部设在加利福尼亚的山景,其研发办公室位于以色列的特拉维夫。Redis Labs 提供了两款 Redis 产品(也包括 Memcached 托管服务):Redis Labs 企业集群(RLEC)和 Redis 云。RLEC 是一款企业集群产品,它将多个 Redis 数据库封装为一个高伸缩性和高可用性的环境,并运行在 Docker 容器中。RLEC 支持多种类型配置,包括只有主节点的单一 Redis 实例的配置,由 Redis 主实例和一到多个 Redis 从实例组成的高可用配置,包含多个主分片节点的 Redis 集群数据库配置,以及最后一个与 Redis 集群相仿的配置。该配置中有多个主分片节点实例,而每个主分片节点有一到多个 Redis 从实例。你可以下载 RLEC 并在自己的环境中使用,也可以购买商业订阅以便在产品环境中使用 RLEC,又或者通过大部分流行的云提供商来使用,例如 AWS、Google、Microsoft 和 Heroku 等。

Redis Labs 通过 Redis Cloud 为组织提供了全面管理的 Redis 托管服务。顾客可以选择使用公有云还是私有云。在本书写作时,Redis Cloud 的定价包括免费的 30MB,36 美元 500MB,71 美元 1GB,338 美元 5GB,由 Redis Labs 负责支持与托管。额外的空间以实际使用付费服务的方式,每 15GB 需要额外支付最高费用的 10%。如果你有针对主从复制、高可用性和集群支持等复杂的设置需求,可以使用 Redis Labs 提供用于管理所有配置和运作支持的增值服务。当然,缺点是这些服务可不便宜,不过这些额外的支持对于你和你的组织来说可能物有所值。

DigitalOcean Redis

在所有提供特定 Redis 托管的云供应商中,DigitalOcean 是最便宜的,并提供了特定 Redis 的虚拟主机“droplet”。这是一种运行 Ubuntu 14.04,预装 Redis,可以使用简单易用的 Web 控制台在几分钟之内开启虚拟机。依据 droplet 的配置差异,最低配的 droplet,即 512MB RAM、1 个 CPU 核心及 1GB SSD 磁盘需要每月 5 美元,最高配的 droplet,即 64GB RAM、20 个 CPU 核心及 640GB 硬盘,需要每月 640 美元。

Digital Ocean 建议要么启用 Redis 配置指令 requirepass(该指令位于 Redis droplet 中/etc/redis/下的 redis.conf 的文件中),要么发送 CONFIG SET requirepass {your-redis-password}到 droplet 上正在运行的 Redis 实例。在同一篇文章中,他们还建议通过更改 CLIEXEC 指令更新/etc/init.d/redis 系统启动脚本:

```
CLIEXEC="/usr/local/bin/redis-cli -a your_redis_password"
```

在该 droplet 中，对 Redis 实例的远程访问通过将 bind 指令设置成 127.0.0.1 的方式禁用了。你需要为运行的 Redis 实例注释掉或者删除 bind 指令，然后通过 ssh 会话连接到 droplet，发送以下命令来重启 Redis 服务：

```
$ sudo service redis restart
```

虽然 DigitalOcean 只是运行 Redis 众多公有云选择中的一个，但是相较于其他选择，Digital Ocean 的 Redis droplet 更简单，可以快速启动并运行一个中小型的 Redis 数据库。

总结

本章介绍了 Linux 容器技术 Docker，它通过提供隔离的、可复制的和快速的应用程序虚拟化而无须启动新的虚拟机的方式，降低管理基于 Redis 的应用程序的难度。我们介绍了从下载到启动官方 Redis Docker 镜像容器实例的步骤，并展示了如何向 Redis 容器中运行的 Redis 发送各种命令行选项。然后，我们介绍了 Docker compose 如何自动化和简化典型应用程序中多容器的部署。最后，我们深入了解了三种运行 Redis 的云托管选项，以及每种选项的优势与劣势。在本书第 9 章中，我们将回归到应用程序设计和开发上来，更深入地探索如何使用 Redis 的发布/订阅功能进行消息通信，以及能够用 Redis 来实现并优化的应用程序间其他类型的功能。

9

任务管理与消息队列

在企业中，Redis 的使用场景多种多样。Redis 对发布/订阅（Pub/Sub）消息通信设计模式的支持使得应用程序能够以快速简单的方式将 Redis 用作消息通信代理。如果 Redis 自带的发布/订阅对于应用程序或者项目来说无法满足需求，那么 Redis 作为更丰富、更多功能的完整消息通信框架也会给予应用程序设计者附加的灵活性。

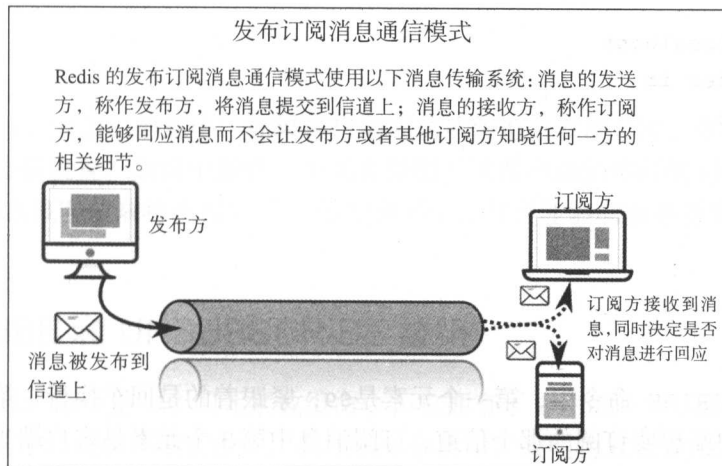
Redis 的发布/订阅模式概述

Redis 的发布/订阅（Publisher/Subscriber，简称 Pub/Sub）是一种消息通信模型，又快又稳定。相较于直接将消息发送给对方这样的处理流程，消息的发布者或者发送方将消息提交到一个或多个信道上，同时消息的接收方或者订阅者订阅到信道上以接收提交到具体信道的所有消息。如果应用程序在设计时更关注竞争条件及可能的消息投递失败的话，那么发布/订阅模式正为此提供了快速的消息通信解决方案。

概念上讲，Redis 的发布/订阅模式与简易信息聚合（RSS）相似。RSS 是网站发布 feed 使用的原子格式，以供客户或者读者消费。不论哪种情况，不管是网站发布者还是消费内容的客户端都不会直接将消息发送给对方。客户端连接到网站并通过 feed 消费网站的出版内容（博客、数据、播客或者其他媒体）。像 RSS、Redis 的发布/订阅模式及其他发布/订阅系统，这种消息通信模式的优势在于针对更动态的网络而言系统具有更佳的伸缩性。

无须为消息的发送方和接收方间的消息路由构建用于管理和协作的代码，大多数发送方和接收方只需从信道上发送和接收消息即可。发布/订阅消息模式的一个不利之处在于发布方的语法较难修正或改变。对于 Redis 来说，发布订阅模式中发布方消息和命令较为稳定且不易更改，同时发布订阅消息格式也很稳定并且分解成了三个或者四个部分，我们会

在下一节讲到：



更明确地讲，Redis 对于发布订阅模式的实现是一种基于主题的消息通信模式，这在 Redis 的术语中称作信道（channel）。除了基于主题的消息通信系统外，其他可选择的发布订阅系统会根据消息的特征和元数据对消息进行路由。下面是一个基于主题的消息通信系统的应用示例，应用程序中的错误处理日志代码会根据应用程序中业务逻辑的分类、I/O 或者网络故障进行错误等级区分，例如 INFO、DEBUG、ALERT 或者 SEVERE，并包含在消息内。如果消息被标记为 ALERT 或者 SEVERE 级别，那么消息的订阅方会以发送邮件的方式进行响应。这类功能虽然没有直接被 Redis 支持，不过仍然能够通过分隔成 INFO、DEBUG、ALERT 或 SEVERE 信道的方式实现，订阅的处理程序在收到来自 ALERT 或 SEVERE 信道的消息时会发送邮件通知。

发布/订阅 RESP 回复

在 Redis 的发布/订阅模式中，消息通信格式是 RESP 数组形式，包含了 3 到 4 个元素。发布/订阅消息通信格式中的第一个元素决定了消息的类型，是以下 4 个 Redis 命令中的一个：SUBSCRIBE、UNSUBSCRIBE、PSUBSCRIBE 和 PUNSUBSCRIBE。

SUBSCRIBE 和 UNSUBSCRIBE RESP 数组

开启两个连接连到 Redis 实例上，第一个采用的是标准的 redis-cli 客户端，第二个用的是 telnet，我们将展示三元素命令 SUBSCRIBE 和 UNSUBSCRIBE 的 RESP 回复：


```
$ telnet localhost 6379
Trying ::1...
Connected to localhost.
Escape character is '^]'.
SUBSCRIBE info
*3
$9
subscribe
$4
info
:1
```

在该 SUBSCRIBE 命令中，第一个元素是\$9，紧跟着的是回车换行。第二个元素是信道 info，即客户端想要订阅的那个信道，订阅消息中第 3 个元素是客户端当前订阅的用于接收消息的信道数量。在我们的 redis-cli 上，我们通过使用 PUBLISH 命令将消息发送到 info 信道上：

```
127.0.0.1:6379> PUBLISH info "Sending a message"
(integer) 1
```

订阅消息的 telnet 客户端收到的 RESP 消息数组如下所示：

```
*3
$7
message
$4
info
$17
Sending a message
```

该三元素数组中，其中第一个元素是 message 关键词，其后是第二个元素信道 info，最后一个元素是真实的消息字符串"Sending a message"。当客户端使用 SUBSCRIBE 命令时，该三元素数组对绝大多数信道来说是 RESP 格式。

现在，我们将对信道 info 发送 UNSUBSCRIBE 命令：

```
UNSUBSCRIBE info
*3
$11
```

```
unsubscribe
```

```
$4
```

```
info
```

```
:0
```

在该 UNSUBSCRIBE 消息中, 如果客户端成功从信道上取消订阅, 那么消息数组中的第二个元素就会被设置, 数组中最后一个元素指明了该客户端仍然订阅的信道数量。如果该数字为 0, 那么该客户端将不再处于发布/订阅模式, 并且能够向服务器发送普通的 Redis 命令。

PSUBSCRIBE 和 UNSUBSCRIBE 数组

PSUBSCRIBE 命令是一个四元素数组命令, 它拥有了 SUBSCRIBE 命令包含的所有字段, 同时额外的字段用作匹配的模式。在 telnet 会话中, 我们将发送 PSUBSCRIBE 命令, 使用星号 “*” 订阅所有以 info 开头的信道:

```
PSUBSCRIBE info*
```

```
*3
```

```
$10
```

```
psubscribe
```

```
$5
```

```
info*
```

```
:1
```

对 PSUBSCRIBE 命令来说, 在 RESP 数组中的第二个元素是 info*, 即用于匹配的模式, 第三个元素是初始信道, 第四个元素是消息体。psubscribe 格式专门用于当客户端使用模式匹配的方式发起订阅时的特殊情况, 关于这一点将在本章下一个小节进行讨论。现在, 我们从 redis-cli 会话发送第二条消息:

```
127.0.0.1:6379>PUBLISH info-1 1
```

```
(integer)1
```

```
*4
```

```
$8
```

```
pmessage
```

```
$5
```

```
info*
```

```
$6
```

```
info-1
```

```

$1
1
*4
$8
pmessage
$5
info*
$6
info-1
$1
1

```

对于 PUNSUBSCRIBE 命令来说,从 Redis 服务器返回的 RESP 数组中包含了 3 个元素,其中之一就是匹配所有信道的模式:

```

PUNSUBSCRIBE info*
*3
$12
punsubscribe
$5
info*
:0

```

使用 redis-cli 进行发布/订阅

Redis 中基本的发布/订阅命令,要从客户端发送带有信道名称的 SUBSCRIBE 命令说起。在本例中,我们在 reid-cli 会话中使用了 status 信道:

```

$ redis/src/redis-cli
127.0.0.1:6379> SUBSCRIBE status
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "status"
3) (integer) 1

```

现在,打开另一个终端窗口(使用窗口工具或者在终端客户端打开单独的页都可以),开启第二个 redis-cli 客户端,并使用 PUBLISH 命令从客户端发送一条消息到 status 信道上:

```
$ redis/src/redis-cli
127.0.0.1:6379> PUBLISH status "Ok, everything working"
(integer) 1
```

切回到之前最初的 redis-cli 窗口中，可以看到如下结果：

```
1) "message"
2) "status"
3) "Ok, everything working"
```

不同于其他客户端，退出第一个 redis-cli 会话需要按下 *Ctrl* + *C* 组合键，退出后会回到 bash。我们将重新启动 redis-cli，并使用 PSUBSCRIBE 命令监测所有以 status 开头的信道：

```
^C
$ redis/src/redis-cli
127.0.0.1:6379> PSUBSCRIBE status*
Reading messages... (press Ctrl-C to quit)
1) "psubscribe"
2) "status*"
3) (integer) 1
```

回到第二个 redis-cli 会话中，也就是我们发送第一条消息的地方。我们将发送一些消息到 status-error、status-alert 和 stats 信道上：

```
127.0.0.1:6379> PUBLISH status-error "Program failed to run"
(integer) 1
127.0.0.1:6379> PUBLISH status-alert "Program approaching maximum memory"
(integer) 1
127.0.0.1:6379> PUBLISH stats "100 Clicks"
(integer) 0
```

在将这些 PUBLISH 命令发送给第一个客户端之后，让我们看看使用 PSUBSCRIBE 命令订阅这些信道客户端上的结果如何：

```
1) "pmessage"
2) "status*"
3) "status-error"
4) "Program failed to run"
1) "pmessage"
```

- 2) "status*"
- 3) "status-alert"
- 4) "Program approaching maximum memory"

因此, 通过使用 PSUBSCRIBE 模式, 监控的 redis-cli 会话收到了来自 status-error 和 status-alert 信道的消息, 但没有收到任何来自 stats 信道的消息。这是因为全局模式没有匹配上该信道。同样值得注意的是, PMESSAGE 响应有 4 个元素, 其中包含了匹配的信道和原始 glob 模式。

值得注意的是, 发布订阅模式的一个重要限制是它的实现不提供消息投递的可靠性, 也就是说, Redis 的发布订阅是发后即忘 (*fire and forget*) 的模式。发布到信道上的消息并不保证能够投递给订阅该信道的客户端。举例来说, 如果发布订阅的客户端监控信道发生了故障, 并在之后重新连接并订阅到信道上, 那么客户端将无法收到这段期间内发往信道的消息。

Redis 发布订阅实战

为了理解 Redis 发布订阅模式在应用程序上下文中如何使用, 我们将观察一个简单的看板 (Kanban) 生产配置, 它包括用于制造一架玩具飞机的三个工作站, 位于北极的一家假想的工厂内: 精灵制造有限公司。看板是一种管理哲学和一整套技术, 是由丰田制造公司在组装汽车过程中首次开发并推广的。在过去的 50 年里, 看板技术的采用展现出了在质量和可靠性方面的显著改进。基于丰田公司在精益生产方面的成功, 围绕看板的实践和哲学在日本传播到了其他制造商, 特别是那些支持丰田公司的供应商和其他公司, 并且传播到了世界各地。到了 2000 年早期, 精益生产已经被广泛接受和采用, 包括那些制造商、服务商、政府和非营利组织。正是因为精益生产, 在有限的预算下提升了产品和服务的质量。



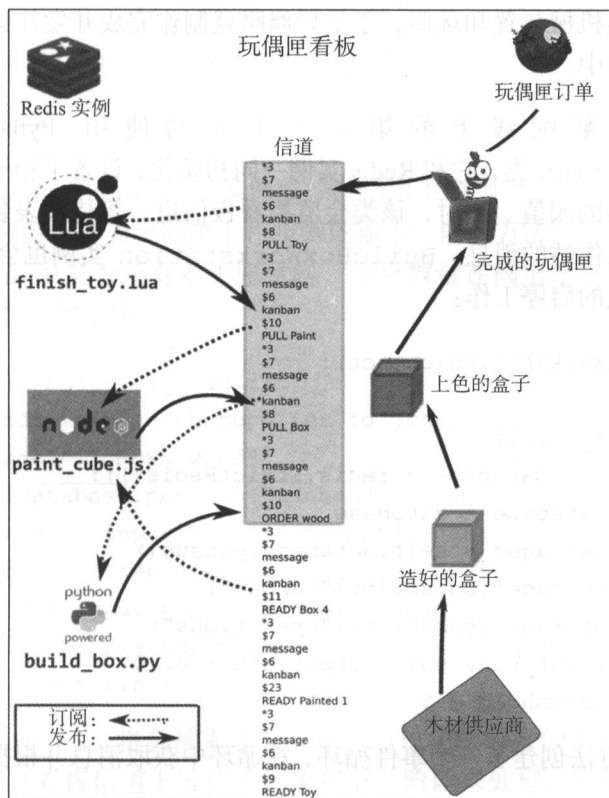
我们使用 PUNSUBSCRIBE * 命令停止监控所有信道, 并将 Redis 客户端切回普通 Redis 模式。

在工厂的传统生产线上, 每个工作站获取前一个工作站的产出物并添加和组装材料, 然后将半成品发往生产线上的下一个工作站。传统制造系统的总体目标是所有工作站的持续吞吐量, 以最少的时间生产出最多的产品, 最终结果是单位成本随着总体产量的提高而降低, 同时带来更多的利润。这种制造方法往往导致自上而下、集中指挥控制的结构, 从而使得企业中的流程紧密耦合。如果其中一个步骤失败, 整个生产流程将戛然而止, 发生

错误的工作站之后的那个工作站将积聚起过量的在制品库存，并且生产线上的下游工作站的制造会逐渐放缓，甚至停止。在精益创业和精益生产文献中将此流程命名为“推动式流程”，相对而言，看板采用的“拉式流程”则是“及时”生产和越来越面向服务化的流程的核心。

在更现实的情况和手工看板操作中，每个生产阶段有物理配色卡片，其上附有相关产品明细，并且关联了一箱子产品生产的每个阶段的原材料。每个箱子含有这一生产阶段所需要消费的部分完成的产品。当箱子中的库存水平快耗尽时，看板卡片和空的箱子就会被发往之前的生产步骤，同时拿回来装满的箱子和看板卡片。流程中的第一步标识了需要发送给供应商的需求订单。

为了模拟三种不同的工作站，我们将以不同的编程语言创建三个应用程序，用来展示 Redis 发布订阅中不同类别的消息通信是如何工作的，以及充当看板工作流程的示例。我们的工厂所要构造的产品是孩子们的“玩偶匣”。它的玩法是当孩子转动控制杆，没过多久，盖子就会打开并蹦出一个小丑的脑袋。



第一个工作站采用 Python 进行发布订阅

在我们虚构的北极产品中的第一个工作站会接收木头原材料，构造一个五面的立方体和一个盖子，并将该产出物传送到生产线的下个步骤。我们将创建一个 Python 类来监控看板信道，当箱子里的材料低于阈值时向供应商请求原材料，建造盒子，然后将箱子和伴随的看板消息一同发往第二个工作站。

在精灵制造厂里的第二个生产工作站有一箱已加工完成并上了色的盒子，一旦在看板信道上接收到“PULL Paint”消息，就会将装满盒子的箱子发送出去。在第二个工作站中，一旦收到消息，加工完成的箱子会被推进到最后一步，同时将“PULL Box”消息发送到看板信道上，向上一个工作站要求一箱粗略构造的木质盒子。在收到来自之前工作站发来的箱子后，第二个工作站将对箱子中的盒子进行上色，在盖子上安装铰链，并在收到来自组装线上最后一个工作站在看板信道上发来“PULL Paint”的消息时，将这箱在制品发送出去。最后一个工作站会接收来自第二个工作站的箱子，里面装满了上了色的盒子。当在看板信道上收到“ORDER Jack-inthe-box”消息时，工作站会从箱子中拿出一个上了色的盒子，为其组装上一个弹出机械装置和玩偶，于是玩偶匣就制作完成并发往运输部门，装入北极雪橇运输公司的袋子中。

我们为生产装配线上的第一个工作站使用 Python 代码编写了 `BuildBoxWorkstation` 类，它和 Redis 实例一同初始化，设置了箱子和用来控制发往供应商请求更多原材料的阈值。同时，该类会监控看板信道，等待代表装满盒子的箱子已经准备好发往第二个工作站的消息。`BuildBoxWorkstation` 实例也会监控 `operations` 信道，监视组装线上的启停工作：

```
class BuildBoxWorkstation(object):

    def __init__(self,
                  database = redis.StrictRedis()):
        self.database = database
        self.messages = self.database.pubsub()
        self.messages.subscribe("kanban")
        self.messages.subscribe("operations")
        self.input_bin, self.output_bin = 0, 0
        self.threshold = 5
```

该类中的 `run` 方法创建了一个事件循环，在循环中获取消息并根据信道和消息进行响

应。为了方便快捷，同时不用响应 SUBSCRIBE、UNSUBSCRIBE、PSUBSCRIBE 和 PUNSUBSCRIBE(如果要构造更稳定的系统，我们会为所有这些类型的消息创建处理程序)，我们将忽略任何数字类型的消息，如下所示：

```
def run(self):
    while 1:
        for item in self.messages.listen():
            channel = item.get('channel')
            message = item.get('data')
            type_of = item.get('type')
            if type(message) == int:
                continue
            message = message.decode()

            if channel == b"kanban":
                if message.startswith("PULL Box"):
                    self.__construct_box__()
            if channel == b"operations":
                if message.startswith("STOP"):
                    return
```

当在 BuildBoxWorkstation 实例监控的信道上收到消息时，要么下单原材料，要么创建好一箱盒子和盖子作为玩偶匣上色步骤的输入。内部方法 `__construct_box__` 首先检测 `input_bin` 中是否还有木头原材料，如果数量在阈值之下，就会在方法中发送 "ORDER wood" 消息，如下所示：

```
def __pull_material__(self):
    # Sends message to supplier to order wood
    message = "ORDER wood"
    self.database.publish("kanban", message)
    print("{} , input_bin={}".format(message, self.input_bin))
    # For convenience we'll just add an order of 5
    # to our input bin in a real operation, an order
    # would be placed, hopefully with a kind supplier API
    self.input_bin += 5
```

下一步，`output_bin` 将会被来自 `input_bin` 中组装好的木质盒子填满，"READY box" 消息会被发送到看板信道上等待生产流程下一阶段来处理，然后将 `input_bin` 和

output_bin 设置为 0。__construct_box__ 代码如下所示：

```
def __construct_box__(self):
    if len(self.input_bin) > 0:
        self.input_bin -= 1
    if len(self.input_bin) <= self.threshold:
        self.__pull_material__()
    # Cuts and assembles box with a lid
    self.output_bin = [i for i in range(1, len(self.input_bin))]
    # Sends Kanban Message to next station
    self.database.publish(
        "kanban",
        "READY Box {}".format(len(self.output_bin))
    )
    # Bins are now empty
    self.input_bin, self.output_bin = 0, 0
```

第二个工作站采用 Node.js 进行发布订阅

我们采用 Node.js 实现第二个工作站。读者可以在本书网站和 GitHub 仓库上找到对应的 paint_cube.js 源文件。paint_cube.js 文件中的第一行导入了 node_redis 模块，创建了两个 Node.js 的 Redis 客户端：

```
var redis = require("redis"),
    client = redis.createClient(),
    client_subscriber = redis.createClient();
```

在创建了两个 Redis 实例之后，paint_cube.js 文件中的第一个函数代表第二个工作站，创建了一个 Javascript 对象，将 Redis 客户端作为对象的数据库，并将两个 Javascript 整数变量 input_bins 和 output_bins 都设置为 0：

```
function PaintCubeWorkstation(redis) {
    var self = this;
    self.database = redis;
    self.input_bins = 0;
    self.output_bins = 0;
```

在设置了这些初始变量之后，PaintCubeWorkstation 实例在每次订阅信道时都会打印一条日志消息：

```

self.database.on("subscribe", function(channel) { console.
log("Subscribed to "+channel); } self.database.subscribe("kanban");
self.database.subscribe("operations");

```

PaintCubeWorkstation 实例中的回调函数用于响应来自 operations 和看板信道上的消息:

```

self.database.on("message", function(channel, message) {
  if(channel === "operations") {
    if(message === "STOP") {
      self.database.unsubscribe();
      self.database.end();
      console.log("Stopping PaintCubeWorkstation");
      process.exit(1);
    }
  }
  if(channel === "kanban") {
    if(message === "PULL Paint") {
      console.log("Output bins " + self.output_bins);
      if(self.output_bins > 0) {
        client.publish("kanban", "READY Painted " +
self.output_bins);
        self.output_bins -= 1;
      } else {
        client.publish("kanban", "PULL Box");
      }
    }
    if(message.indexOf("READY Box") === 0){
      self.input_bins += 1;
      console.log("Input bin size " + self.input_bins)
      for(i = 0; i<=self.input_bins; i++) {
        // adds hinges and paints each box and adds to output bin
        console.log("\tAdds hinges and paints " + i);
        self.output_bins += 1;
      }
      client.publish("kanban", "READY Painted " +
self.output_bins);
    }
  }
}

```

```

    }
}

```

paint_cube.js 文件中的最后几行实例化了 PaintCubeWorkstation 对象,对那些受到监控的信道,使用回调函数来处理发送至这些信道上的所有消息。

第三个工作站使用 Lua 客户端进行发布订阅

对于为客户组装玩偶匣的最后的工作站来说,我们将采用两个 Lua 脚本,其中一个服务器端 Lua 脚本,用于在 redis-cli 上进行调用,另一个客户端 Lua 脚本用来订阅并响应消息。像 BuildBoxWorkstation Python 代码和 aintCubeWorkstation Node.js 代码那样,客户端 Lua 脚本会响应玩具订单,它首先使用在输出箱内的现存玩偶匣,并在存货不足时,向看板信道发送消息,用来向第二个工作站请求新的部分完成的玩具。

为了搭建本地环境来支持客户端 Lua 脚本,我们首先需要从 <http://www.lua.org/> 上安装最新版本的 Lua,再安装 LuaRocks 包管理器,可以在 <https://luarocks.org/> 上找到。下一步,我们需要使用下列命令安装 redis-lua 客户端(假设你使用的是 Linux 或者 Mac),redis-lua 客户端可以在 <https://github.com/nrk/redis-lua> 上找到:

```
$ sudo luarocks install redis-lua
```

安装了 Lua 的 Redis 客户端之后,我们就可以创建订阅服务来处理和响应来自看板和 operations 信道的消息。Lua 脚本 finish_toy.lua 包含了用于监听这些信道的函数,并基于收到的"ORDER Toy"消息调用对应的服务器端 Lua 脚本向看板信道发送消息。不像之前两个工作站的代码那样,最后一个工作站只有一个 Lua 表格类型的单一箱子,详情请参考 finish_toy.lua。

在 Lua 客户端脚本的开头部分,我们导入并设置了两个 Redis 客户端,然后我们创建了 Lua 变量用来存放待投递的已完成的玩偶匣,以及 Lua 表格 channels:

```

local redis = require 'redis'
local client = redis.connect('127.0.0.1', 6379)
local publisher = redis.connect('127.0.0.1', 6379)
local toys = 0
local channels = {"kanban", "operations" }

```

接下来,我们定义了两个函数 build 和 deliver,当监听的信道上发来"PULL toy"或者"READY Painted"的玩偶匣消息时会被 main 函数调用。

build 函数打印一条消息，并将全局变量 toys 增加了 1:

```
function build ()
    print("Building Toy "..toys)
    toys = toys + 1
end
```

deliver 函数将消息发送至看板信道，打印一段话，并将变量 toys 减 1:

```
function deliver ()
    publisher:publish("kanban", "READY Toy")
    print("Toy delivered")
    toys = toys - 1
end
```

main 函数创建了 pubsub 循环，检查消息的值，并根据消息的信道和内容来决定接下来的步骤。当在 operations 信道上接收到 STOP 消息时，程序会退出 pubsub 模式并返回一个布尔值来终止操作。依据来自 kanban 信道的消息内容，要么将玩具投递出去，要么将"PULL Paint"消息发送到 kanban 信道上。

最后，当 PaintCubeWorkstation 将"READY Painted"消息发送到看板信道时，就会调用 Lua 脚本的 build 和 deliver 函数，如下所示:

```
for msg, abort in client:pubsub({ subscribe = channels}) do
    if msg.kind == 'message' then
        if msg.channel == "operations" then
            if msg.payload == "STOP" then
                print("Stopping Finish Toy")
                abort()
            end
        elseif msg.channel == "kanban" then
            if msg.payload == "PULL Toy" then
                if toys > 0 then
                    deliver()
                else
                    publisher:publish("kanban", "PULL Paint")
                end
            elseif msg.payload == "READY Painted" then
                build()
            end
        end
    end
end
```

```

        deliver()
    end
end
end
end
end

```

在介绍完这三个工作站的代码之后，我们通过在命令行终端会话上启动三个工作站的代码，来模拟运行玩偶匣看板流程。我们首先启动 redis-cli 会话：

```

127.0.0.1:6379> PUBLISH kanban "PULL Toy"
(integer) 3

```

"PULL Toy"消息触发了 Lua 脚本中的操作，并向命令行打印了以下内容：

```

$ lua finish_toy.lua
Final Assemble Toy Workstation
Pull Paint box
Building Toy 0
Toy delivered

```

监控第二个工作站，我们会看到 Node.js PaintCubeWorkstation 对象的输出结果如下：

```

$ node paint_cube.js
In Paint Cube Application
Subscribed to kanban
Subscribed to operations
Output bins 0
Input bin size 1
    Adds hinges and paints 0
    Adds hinges and paints 1

```

第一个工作站响应了来自 Paint Cube 工作站的"PULL Box"消息，因而 Python 模块 build_box 会在终端窗口输出以下内容：

```

$ python3 build_box.py
Running BuildBoxWorkstation
ORDER wood, input_bin=0
READY Box 4
READY Box 3

```

READY Box 2

现在，我们将向 operations 信道发送 "STOP" 消息。所有工作站都订阅了该信道。这会导致所有工作站都停止工作：

```
127.0.0.1:6379> PUBLISH operations "STOP"
(integer) 3
```

build_box.BuildBoxWorkstation 实例在响应 STOP 消息后输出了以下内容：

```
Stopping
$
```

Node.js 的 PaintCubeWorkstation 在响应来自 operations 信道的 STOP 消息时，会向终端打印一条消息，并发送整数 1 来调用 process.exit，以终止脚本的执行并回到 bash shell。

```
Stopping PaintCubeWorkstation
$
```

finish_toy.lua 脚本的 main 函数打印了一段文本，并将控制权交还给 bash shell：

```
Stopping Finish Toy
$
```

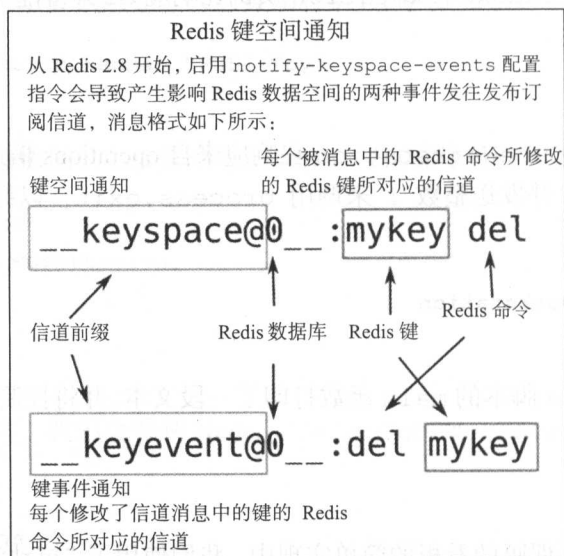
在这个用来组装玩偶匣的看板的简单实现中，我们使用了“拉动流程”替代传统的“推动流程”构造每件玩具。为了模拟 Redis 发布订阅消息通信是如何使用不同的应用程序系统的，我们分别使用了 Python、Node.js 和 Lua 创建三种不同的脚本，并展示了采用各自编程语言的 Redis 客户端是如何仅使用 Redis 发布订阅消息通信框架和相关命令进行互操作的。

Redis 键空间通知

使用 Redis 时的一个常见用例场景是，应用程序可以对存储在一些特定键上的值的变化进行响应。幸运的是，Redis 从 2.8 版本开始提供了一种机制，客户端代码可以通过订阅到信道上来监控与数据相关的事件。这种称为键空间通知的功能可以用来监控针对键的事件，例如更改指定键的命令，或者是收到诸如 HSET 等特定命令的键，或者是由于 EXPIRE 命令的关系而要被删除的键。使用 Redis 的发布订阅功能可以使现有的实现了发布订阅的 Redis 客户端使用键空间通知来响应 Redis 数据的变化。

当发出一个命令触发了 Redis 键空间通知时, 监控的客户端可以响应两种事件: 第一种叫作键空间通知, 第二种叫键事件通知。对于键空间通知事件来说, 当 Redis 哈希 `your_key` 键中的字段值被更改时, 消息会被发往信道 `__keyspace@0__:your_key hset`。

与此同时, 另一条消息会作为键事件通知发往 `keyevent@0__:hset your_key` 信道。在 Redis 中, 所有监控 HSET 命令的客户端都会收到:



默认情况下, Redis 键空间通知是禁用的, 这是因为虽然该功能并不需要对内存进行密集操作, 但是它仍然需要额外的 CPU 资源。为了启用键空间通知, 可以通过修改并启用 `redis.conf` 中的 `notify-keyspace-events` 配置指令, 或者通过 `CONFIG SET Redis` 命令启用。 `notify-keyspace-events` 指令接收许多参数, 用来决定信道的类型 (键空间还是键事件) 和发送到信道上的内容:

- K 参数代表了所有键空间事件。
- E 参数代表了所有键事件。必须设置两者中的至少一个, 以启用通知, 不然的话所有信道都被禁用。

监控的命令类型由以下参数决定:

- \$ 代表字符串
- l 代表列表

- s 代表集合
- h 代表哈希
- z 代表有序集合

最后, x 参数代表键过期的消息。e 参数代表由于触碰了运行时 Redis 实例的最大内存限制而导致键被驱逐的消息。

回到之前的关联数据片段服务器项目上, 我们在该项目的最新迭代中采用了 Redis 键空间通知。最新版本的关联数据片段服务器没有使用 KEYS 或者 SCAN 来匹配三元组模式, 而是采用了哈希来表示三元组。但是缺少一种从这些哈希中自动化删除字段的方法, 在元素过期时通过 Redis 缓存的 LRU 策略被驱逐出 Redis 实例。虽然人们提出了有关为哈希设置单个字段过期这一功能, 但是现阶段的 Redis 并没有该功能, 将来也没有计划实现。不过, 我们可以通过采用 Lua 脚本和键空间通知来实现这一功能。

为了启用键空间通知, 我们修改了 redis.conf 文件, 使得 notify-keyspace-events 配置指令包含参数 AE, 用来监控所有过期的键事件。

在关联数据片段服务器的主缓存模块 cache/__init__.py 中, 新的函数 remove_expired 为了过期键来监控键事件信道 __keyevent@__:。作为缓存服务器, 关联数据片段服务器为存储在 Redis 中的每个主语、谓语和宾语的 SHA1 哈希都设置了过期时间限制。当特定的键过期时, 我们将首先检查任何关联的谓语宾语、主语谓语和主语宾语哈希, 迭代这些哈希, 并从其他键的关联哈希中移除过期的哈希摘要。

remove_expired 函数调用了三个函数 remove_subject、remove_object 和 remove_predicate, 用来删除过期的二级键, 以及缓存中其他二级键 (这依赖于三元组在 Redis 缓存中采用的 Redis 数据结构策略)。Python 函数 remove_expired 如下所示:

```
def remove_expired(**kwargs):
    datastore = kwargs.get("datastore", redis.StrictRedis())
    strategy = kwargs.get("strategy", "string")
    database = kwargs.get('db', 0)
    if strategy.startswith('string'):
        return
    expired_key_notification = "__keyevent@{}__:expired"
    expired_pubsub = datastore.pubsub()
    expired_pubsub.subscribe(expired_key_notification)
    for item in expired_pubsub.listen():
```



```

    sha1 = item.get("data")
    transaction = datastore.pipeline(transaction=True)
    remove_subject(sha1, transaction, datastore)
    remove_predicate(sha1, transaction, datastore)
    remove_object(sha1, transaction, datastore)
    transaction.execute()

```

remove_subject、remove_object 和 remove_predicate 函数在结构和目的上是相似的，以下 remove_subject 函数展示了用来移除关联数据片段服务器的 Redis 缓存中表示三元组的主语、谓语或者宾语中任意成员或者关联关键字段的方法。

```

def remove_subject(
    digest,
    transaction,
    datastore=redis.StrictRedis()):
    subject_key = "{}:pred-obj".format(digest)
    if not datastore.exists(subject_key):
        return
    for row in datastore.smembers(subject_key):
        predicate, object_ = row.split(":")
        pred_subj_obj = "{}:subj-obj".format(predicate)
        if datastore.exists(pred_subj_obj):
            transaction.srem(pred_subj_obj,
                             "{}:{}".format(digest, object_))
        obj_subj_pred = "{}:subj-pred".format(object_)
        if datastore.exists(obj_subj_pred):
            transaction.srem(
                obj_subj_pred,
                "{}:{}".format(digest, predicate))
    transaction.delete(subject_key)

```

举例来说，<http://catalog.coloradocollege.edu/abde34> URL 的 SHA1 哈希为 1dac26e30da98f3b64ce7e0e6de9704e18deefd1，它拥有下列 Redis 哈希 1dac26e30da98f3b64ce7e0e6de9704e18deefd1:pred-obj 用来存储所有谓语和宾语的 SHA1 哈希，共同组成完整的三元组。当 1dac26e30da98f3b64ce7e0e6de9704e18deefd1 过期时，消息会被发送给信道，函数会查看哈希中的字段，并移除在 Redis 缓存上的集合或者哈希中所有对 1dac26e30da98f3b64ce7e0e6de9704e18deefd1 的引用。

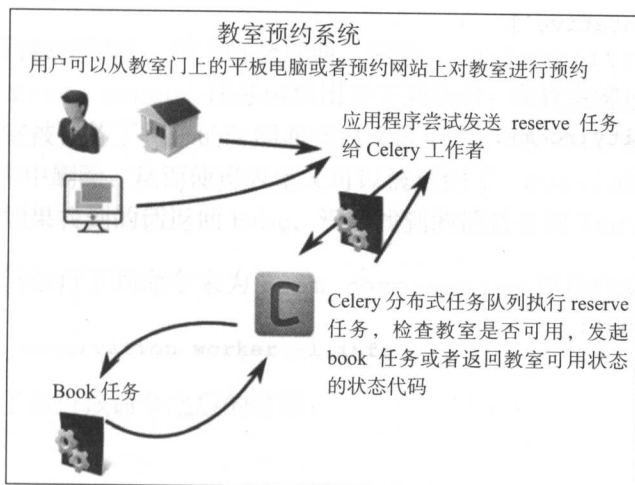
使用 Redis 和 Celery 进行任务管理

Celery 作为一个异步任务队列项目，基于分布式消息传递并允许独立任务的执行。详情请访问 <http://www.celeryproject.org>。你可以在 Celery 中指定不同的消息代理服务器，让 Redis 成为消息代理服务器之一。在 Celery 的基本用例场景中，基于 Celery 的应用程序负责处理创建任务及管理用于响应那些任务的工作者。

为了展示如何在简单的应用程序中使用 Celery，我们将实现的应用程序基于如下场景：一个小型学区中有一所高校想要一个教室预约系统来举办课外活动。虽然有许多方法来实现这个简单的预约系统，但是我们将为这所高校构建基于 Python 的系统，使用 Celery 和 Redis 来管理教室预约。每间可被预约的教室可以切换到下列状态：

- Cancelled
- Confirmed
- Denied
- Mediated
- Tentative

当预约任务从其他应用程序发送到教室预约系统时，返回的状态依赖于多种因素。如果该教室可以被预定，依据用户或者应用程序的权限会自动返回状态 Tentative 或者 Confirmed。如果教室已经被预定，则用户将会收到 Denied 状态。如果用户或者应用程序没有权限来预约特定的教室，那么就会返回 Mediated 状态。在典型的学校设置中，老师可以预约教室，学生可以暂时从学校网站上预约教室。



每间教室的门上都附有平板电脑来展示预约状态，以及接下来这间教室有哪些预约。采用像 Celery 这样的任务与消息框架的架构的目的在于，可以轻松添加更多的教室以应对预约请求，同时展示它们当前的状态。

我们将为项目创建新的目录，并创建三个文件 `init__.py`、`backend.py` 和 `tasks.py`。`__init__.py` 可以为空，存在于目录中是为了能以 Python 模块的方式使用我们的应用程序。在 `backend.py` 文件中，我们将导入 `celery` 模块，并使用 Redis 代理创建 Celery 应用程序：

```
from celery import Celery

app = Celery("room_reservation",
             broker="redis://localhost",
             backend="redis://localhost/1")
```

Celery 应用程序将 Redis 实例用作消息代理和后台结果存储。`tasks.py` 代码文件中导入了 Celery 应用程序，并采用 `task` 装饰器将 `availability`、`reserve`、`book`、`cancel`、`search` 和 `room` 函数指定为 Celery 任务：

```
from celery import app

STATUS = ['Cancelled',
          'Confirmed',
          'Denied',
          'Mediated',
          'Tentative']
```

```
@app.task
def availability(room):
```

```
    .
    .
    .
```

```
@app.task
def book(status):
```

```
    .
    .
    .
```

```
@app.task
```



```
[2015-10-24 11:52:14,647: WARNING/MainProcess]
/usr/local/lib/python3.4/dist-packages/celery/apps/worker.py:161: CDeprecationWarning:
Starting from version 3.2 Celery will refuse to accept pickle by default.
```

The pickle serializer is a security concern as it may give attackers the ability to execute any command. It's important to secure your broker from unauthorized access when using pickle, so we think that enabling pickle should require a deliberate action and not be the default choice.

If you depend on pickle then you should set a setting to disable this warning and to be sure that everything will continue working when you upgrade to Celery 3.2.:

```
CELERY_ACCEPT_CONTENT = ['pickle', 'json', 'msgpack', 'yaml']
```

You must only enable the serializers that you will actually use.

```
warnings.warn(CDeprecationWarning(W_PICKLE_DEPRECATED))

----- celery@LibrarySystems v3.1.19 (Cipater)
---- **** ----
--- * *** * -- Linux-3.13.0-66-generic-x86_64-with-Ubuntu-14.04-trusty
-- * - **** ---
- ** ----- [config]
- ** ----- .> app:      room_reservation:0x7fe97203eeb8
- ** ----- .> transport: redis://localhost:6379//
- ** ----- .> results:  redis://localhost
- *** --- * --- .> concurrency: 1 (prefork)
-- ***** ---
-- ***** --- [queues]
----- .> celery      exchange=celery(direct) key=celery

[tasks]

[2015-10-24 11:52:14,684: INFO/MainProcess] Connected to redis://localhost:6379//
[2015-10-24 11:52:14,694: INFO/MainProcess] mingle: searching for neighbors
[2015-10-24 11:52:15,700: INFO/MainProcess] mingle: all alone
[2015-10-24 11:52:15,715: WARNING/MainProcess] celery@LibrarySystems ready.
```

举例来说,为了检测教室 101 是否可用,我们将启动 Python shell,导入 availability 任务并立即运行:

```
$ python3
Python 3.4.3 (default, Oct 14 2015, 20:28:29)
[GCC 4.8.4] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from room_reservation.tasks import availability
```

```
>>> result = availability.delay("room-101")
```

availability 任务被立即执行, 结果如下所示:

```
[2015-10-24 12:41:18,637: INFO/MainProcess] Received task: room_reservation.
```

```
tasks.availability[abc10c51-beb3-4e8b-abc5-21dda4dd7ddf]
```

```
[2015-10-24 12:41:18,641: INFO/MainProcess] Task room_reservation.
```

```
tasks.availability[abc10c51-beb3-4e8b-abc5-21dda4dd7ddf] succeeded in
```

```
0.0029513250046875328s: -1
```

客户端应用程序以类似的方式发送 reserve 任务, 然后使用 availability 和 book 任务完成基于 Celery 和 Redis 的分布式异步的教室预定。

GIS 和 RestMQ

地理数据结构和命令已经被添加到 Redis 3.2 分支上, 并为向基于 Redis 的项目添加 GIS 功能开启了新的大门。在我们深入研究如何使用 Redis 构建基于 GIS 的消息通信系统前, 我们先来探索这些新的基于地理的命令的基本操作。

目前为止, 地理命令和功能只在 Redis 3.2 版本才可用。为了使用这些命令, 你需要下载 3.2 版本的 Redis, 再进行编译使用。

基于地理的命令是通过采用一种叫作 geohashing 的技术实现的。该技术是一种纬度/经度编码系统, 它将空间层次结构构造成网格上的桶。geohash 算法是由 Gustavo Niemeyer 为 <http://geohash.org> 网站服务创建的。geohash 的构造允许附近地理位置共享同样的字符串。由于定义了哈希字符串, 位置的精度逐渐限制到一个点上。因为附近位置共享前缀, geohashing 允许用户通过移除 geohash 右边的字符来逐渐扩大搜索范围。

用来支持地理应用的前几个 Redis 命令是 GEOADD 和 GEODIST。命令接收一个键和一个或者多个由纬度、经度和元素名称组成的三元组。位置的 geohash 存储在 Redis 的有序集合中, 因此每个位置需要一个元素名称来表示。为了使用 GEOADD 命令, 我们首先启动 Redis 3.2 实例, 然后在另一个窗口启动 redis-cli 程序来连接数据库。我们将构建一个滑雪天气消息通信应用程序, 将当前滑雪斜坡环境发送到滑雪者或者滑雪运动员的手机上。我们将使用 GEOADD 创建科罗拉多山的滑雪场, 包含美国科罗拉多的四个滑雪胜地的经度和

纬度。

```
127.0.0.1:6379> GEOADD colorado_ski_mountains -106.926982 38.905476
"Crested Butte"
(integer) 1
127.0.0.1:6379> GEOADD colorado_ski_mountains -106.3381 38.502855
"Monarch Mountain"
(integer) 1
127.0.0.1:6379> GEOADD colorado_ski_mountains -106.822146 39.165098
"Aspen Mountain"
(integer) 1
127.0.0.1:6379> GEOADD colorado_ski_mountains -106.355999 39.605234 "Vail
Mountain"
(integer) 1
```

为了使用 Redis 键 `colorado_ski_mountains`，我们可以使用 `GEODIST` 命令计算 Crested Butte 滑雪山和 Aspen 山之间的距离，如下所示：

```
127.0.0.1:6379> GEODIST colorado_ski_mountains "Crested Butte" "Aspen
Mountain"
"30263.881549595"
```

我们也可以使用下列命令返回以千米或者英里为单位的距离：

```
127.0.0.1:6379> GEODIST colorado_ski_mountains "Crested Butte" "Aspen
Mountain" km
"30.263881549595002"
127.0.0.1:6379> GEODIST colorado_ski_mountains "Crested Butte, Colorado"
"Aspen Mountain" mi
"18.80515090011744"
```

`GEOHASH` 命令返回一个或者多个元素的 geohash。由于 Crested Butte 和 Aspen 山相距比较近，我们可以从 `GEOHASH` 命令执行的结果中注意到，它们两者最左边的字符 "9w" 是一样的：

```
127.0.0.1:6379> GEOHASH colorado_ski_mountains "Crested Butte" "Aspen
Mountain"
1) "9wgvqfd0ud0"
2) "9wuncly3px0"
```

GEOPOS 命令返回 GEO 有序集合中元素的经度和纬度。下面展示的是 Vail 和 Monarch 山的经度和纬度：

```
127.0.0.1:6379> GEOPOS colorado_ski_mountains "Vail Mountain"
1) 1) "-106.35600060224533"
   2) "39.605234833330236"
127.0.0.1:6379> GEOPOS colorado_ski_mountains "Monarch Mountain"
1) 1) "-106.33809953927994"
   2) "38.502854184479808"
```

GEORADIUS 命令接收 geohash 键，经纬度的中心位置及米、千米、英尺或者英里为单位的半径，并返回离中心最近的位置。GEORADIUS 包括下列额外选项，WITHCOORD 选项用于获取每个位置的经度和纬度，WITHDIST 选项给出了位置和中心半径的距离，还有 WITHHASH 选项给出了原始 geohash 编码的集合分值。我们使用美国科罗拉多州中心的经度和纬度，在 redis-cli 会话中展示 GEORADIUS 命令的不同选项：

```
127.0.0.1:6379> GEORADIUS colorado_ski_mountains -105.692242 38.875350
100 km
1) "Vail Mountain"
2) "Monarch Moutain"
127.0.0.1:6379> GEORADIUS colorado_ski_mountains -105.692242 38.875350
100 km WITHCOORD
1) 1) "Vail Mountain"
   2) 1) "-106.35600060224533"
      2) "39.605234833330236"
2) 1) "Monarch Moutain"
   2) 1) "-106.33809953927994"
      2) "38.502854184479808"
127.0.0.1:6379> GEORADIUS colorado_ski_mountains -105.692242 38.875350
100 km WITHHASH
1) 1) "Vail Mountain"
   2) (integer) 1396750187740657
2) 1) "Monarch Moutain"
   2) (integer) 1396482048060275
```


使用 RestMQ 进行任务管理

在了解了这些 Redis 地理命令之后，现在我们通过采用 RestMQ 添加任务管理功能。RestMQ 是一个开源项目，用 Python 实现了消息队列。读者可以从 <https://github.com/gleicon/restmq> 上进行下载并安装。RestMQ 能以 Docker 容器的方式运行。在安装运行 RestMQ 和 Redis 之前，我们先克隆代码仓库：

```
$ git clone https://github.com/gleicon/restmq.git
Cloning into 'restmq'...
remote: Counting objects: 796, done.
remote: Total 796 (delta 0), reused 0 (delta 0), pack-reused 796
Receiving objects: 100% (796/796), 241.22 KiB | 0 bytes/s, done.
Resolving deltas: 100% (426/426), done.
Checking connectivity... done.
```

我们想要运行 Redis 的 3.2 分支以便使用 GIS 命令，因此我们将修改 restmq 的 Dockerfile，下载并解压 redis-tar 文件，编译 redis-server，并将其复制到 /usr/bin/redis-server 中，并将该路径用作 Dockerfile 中的 RUN 行：

```
RUN apt-get install -y wget && \ wget http://download.redis.io/releases/
redis-3.2.0.tar.gz && \
    tar xzvf redis-3.2.0.tar.gz &&\
    cd redis-3.2.0 && \
    make && \
    cp src/redis-server /usr/bin/redis-server
```

因为 Redis 3.2 默认以受保护模式运行，所以我们需要修改 restmq/dockerfiles/supervisor/redis.conf 文件并在第二行添加以下内容：

```
command=/usr/bin/redis-server --protected-mode no
```

我们将发送下列 Docker 命令来构建 RestMQ 镜像，然后在端口 8888 上运行 RestMQ，同时在默认端口 6379 上运行 Redis 实例：

```
$ docker build -t restmq .
$ docker run --rm -p 6379:6379 -p 8888:8888 restmq
```

RestMQ 服务器运行在端口 8888 上，并接收 HTTP 的 GET、POST 和 DELETE 方法。这些 HTTP 动作被定义为在 <http://localhost:8888/q/<queue>> 上的 REST 服

务, 其中:

- GET 请求将从 RestMQ 队列移除对象
- POST 请求将向 RestMQ 队列插入对象
- DELETE 请求将清除队列

为了展示我们的天气应用程序会监控天气状况并且在方圆 100km 内有天气状况发生时警告所有的滑雪胜地, 我们将编写下列 Python 函数 `monitor_weather`, 它会轮询 RestMQ 队列, 将纬度和经度从天气事件中提取出来, 运行 Redis 命令, 然后将告警信息发布到 RestMQ 队列以通知每个滑雪胜地:

```
def monitor_weather(
    base_url,
    datastore=redis.StrictRedis()):
    channel_url = "{}c/monitor".format(base_url)
    monitor_resp = requests.get(channel_url, stream=True)
    line_buffer = str()
    for char in monitor_resp.iter_content():
        line_buffer += char.decode()
        if line_buffer.endswith('\r\n'):
            line = line_buffer[0:-2]
            if line.startswith('null'):
                break
            message = json.loads(line)
            result = json.loads(message.get('value'))
            if 'location' in result:
                location = result.get('location')
                alert = result.get('event')
                in_ski_area = datastore.execute_command(
                    "GEORADIUS",
                    "colorado_ski_mountains",
                    location.get('longitude'),
                    location.get('latitude'),
                    100,
                    "km",
                    "WITHHASH")
                # Goes through each resort and add a weather alert
```

```
# to a queue resort's hash value
for row in in_ski_area:
    queue_url = "{} /q/{}".format(base_url, row[1])
    alert_result = requests.post(queue_url,
                                  data={"value": alert})
```

我们可以从该函数中看到怎样以相当简单的方式将 Redis 用作 GIS 计算和消息队列服务器,从而实现 GIS 应用程序的。为了了解使用 `monitor_weather` 的详细内容,请读者查阅 `app.py` 模块文档。在下一节中,我们将探索另一个消息通信选项,它不使用 Redis 服务器,但是采用 RESP 及 Redis 设计模式。

使用 Redis 技术进行消息通信

在分布式系统设计中,一种流行的使用模式是实现一个消息队列,充当生产者和消费者之间通信的中介平台。生产者和消费者不必和消息通信服务器运行在相同的机器上,消息通信服务器也可能不使用 Redis 自身的发布订阅命令。

使用 Disque 进行消息通信

在 2015 年早些时候,Salvatore Sanfilippo 宣告并发布了 Disque 的首个 alpha 版本。这是一个全新的分布式消息代理项目,可以在 <https://github.com/antirez/disque> 上找到源代码。Disque 基于 Redis 协议,但却不是真正意义上的 Redis 服务器。Redis 客户端可以与 Disque 通信并使用 Disque。不过,已经有许多最流行的编程语言的 Disque 客户端发布了,并且可以从 Disque 的 GitHub 主页上获取。

运行 Disque 的方式和 Redis 的如出一辙。首先,打开一个终端窗口,可以使用 Git 克隆 Disque 仓库或者从 <https://github.com/antirez/disque/archive/master.zip> 上下载代码库的 ZIP 文件:

```
$ git clone https://github.com/antirez/disque.git
Cloning into 'disque'...
remote: Counting objects: 2565, done.
remote: Total 2565 (delta 0), reused 0 (delta 0), pack-reused 2565
Receiving objects: 100% (2565/2565), 1.38 MiB | 441.00 KiB/s, done.
Resolving deltas: 100% (1617/1617), done.
Checking connectivity... done.
```

下一步，我们将目录切换到 Disque，然后运行 make 命令：

```
$ cd disque
$ make
```

在编译完 Disque 之后，我们将启动 Disque，并开始研究其是否有能力作为分布式作业或者任务队列来应对一种假想的场景：我们需要一种太阳系内的通信系统，用于地球、月亮、火星、小行星和彗星之间的通信。Disque 的设计模式类似于 Redis，因而我们会创建节点的集群，这些节点运行在不同的端口，每个节点使用 disque.conf 拷贝。为了开始对 Disque 的实验，我们将创建一个项目目录并复制四份 disque.conf：

```
$ mkdir solar-com
$ cd solar-com
$ cp ~/disque/disque.conf .
```

运行 disque-server 来启动一个简单的 Disque 实例（如果你对以集群方式运行 Disque 不感兴趣，可以看看 Disque 网站上的文档）：

```
$ ~/disque/src/disque-server disque.conf
```

现在，我们将创建两条消息并使用 Disque 客户端的 ADDJOB 命令发送至 Earth 队列。该命令接收三个参数，第一个参数是文本字符串用于表示队列名称，第二个参数是作业的名称，最后一个参数是以毫秒为单位的超时时间：

```
$ ~/disque/src/disque
127.0.0.1:7711> ADDJOB Earth "Get latest news" 0
DI11e16675a292b568ad0e7a01ecc51aeceb53bd9105a0SQ
```

通过使用 Disque 命令 GETJOB 和 FROM 关键字，我们可以从 Earth 队列获取任意作业，每次获得一条：

```
127.0.0.1:7711> GETJOB FROM Earth
1) 1) "Earth"
   2) "DI11e16675a292b568ad0e7a01ecc51aeceb53bd9105a0SQ"
   3) "Get latest news"
```

现在，我们将使用 ADDJOB 发送第二条消息到 Moon 队列，并使用 GETJOB 从 Moon 队列获取消息：

```
127.0.0.1:7711> ADDJOB Moon "Get moon rock sample" 0
```

```
DI11e166757cd789784008299f7a393dbbdd36e14d05a0SQ127.0.0.1:7711> GETJOB  
FROM Moon
```

```
1) 1) "Moon"
```

```
2) "DI11e166757cd789784008299f7a393dbbdd36e14d05a0SQ"
```

```
3) "Get moon rock sample"
```

我们可以看到这里的输出和 redis-cli 会话类似, Disque 采用 Redis 的命名规则及 Redis 成功地执行模型并提供比 Redis 更丰富的消息通信选项。

总结

在本章中,我们详细讲解了 Redis 对于发布订阅消息通信模式的支持,客户端可以通过发送 SUBSCRIBE 或者 glob 风格模式匹配变体的 Redis 命令 PSUBSCRIBE 来进行订阅。当处于发布订阅模式时,客户端将不能使用其他 Redis 命令。客户端监控一个或者多个信道,等待消息的到达。其他客户端可以使用 PUBLISH 命令向信道推送消息,同时所有订阅信道的客户端,不管是直接订阅还是使用模式匹配的方式,都会收到消息。为了展示如何使用三种不同的编程语言和客户端进行 Redis 的发布订阅,我们构造了一个简化的看板生产流程,用来为虚构的北极公司制造玩偶匣玩具。之后我们研究了两种任务与消息通信框架,即 Celery 和 RestMQ,利用 Redis 为客户端应用程序实现更丰富、更健壮的消息通信。这些示例包括了学校教室预约系统,使用了 Redis 的全新 GIS 命令和 RestMQ 的地理应用。最后,我们研究了由 Redis 之父 Salvatore Sanfilippo 创作的相对较新的项目 Disque,它是用 C 语言编写的非阻塞网络服务器,采用 RESP (因为 Redis 标准客户端可以进行连接并使用) 管理一个分布式、基于内存的消息代理。

最后一章的内容来自于本书其他章节中的知识和技能,展示了 Redis 如何应用于大多数 ETL 工作流之中,并在现代企业中的“大数据”分析与管理中扮演了至关重要的角色。

10

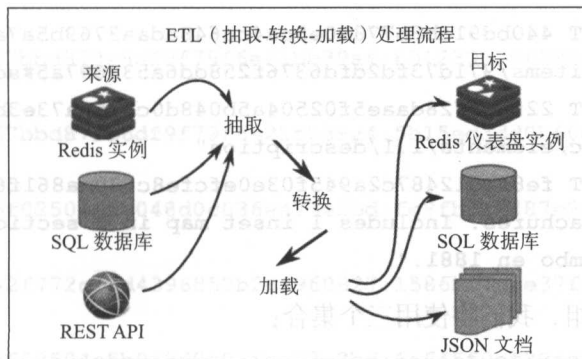
信息流的测量与管理

本章将专注于 Redis 在信息捕获和数据分析中扮演的角色，为企业带来真正有价值的信息。首先，我们将详细介绍如何通过大量数据插入（mass insertion）和其他技术方法将大量的数据进行抽取、转换并加载到 Redis 中，并给出相关示例。接下来，我们将探究基于安全方面的考虑是如何影响管理决策的，以及基于 Web 的仪表盘用来展示 Redis 的运行时统计数据。最后，我们还将研究如何将 Redis 用于机器学习技术，例如朴素贝叶斯和线性回归。

基于 Redis 的 ETL 方案

Redis 的灵活性和速度使得它非常适合用于同类数据及复杂异构数据源的抽取、转换和加载（ETL）过程。这在企业中已变得越来越普遍。

不像那些昂贵专用的 ETL 系统，Redis 的开源模型提供了甚至连这些商用系统都缺少的功能，给中小型企业带来更多机遇来改进不断增长、越来越复杂的数据的处理流程。



对绝大多数 Redis 客户端,甚至是那些支持事务和命令流水线的 Redis 客户端来说,导入数据是非常缓慢且低效的。这是由 Redis 服务器和 Redis 客户端之间往返的写入和响应所造成的。大多数客户端不支持非阻塞 I/O 模型,经常不能以最大化客户端和服务端间吞吐量的方式解析从 Redis 返回的响应。在本书的大多数示例中,我们在往运行中的 Redis 实例导入大量数据时,忽略了这些性能影响。就如官方 Redis 文档提到的关于大量数据插入,请参考附录中来源的第 10 章:信息流的测量与管理,以最快的方式往 Redis 里导入数据的推荐方式是生成包含原始 Redis 协议 (RESP) 的文本文件。之后可以在 Netcat 或者 redis-cli 上使用特殊的流水线模式导入该 RESP 文本文件。

为了对比使用 Redis 客户端在一个事务中发送多条命令和使用管道方法将原生 RESP 插入 Redis 中,我们将通过修改现有的关联数据片段服务器项目中的代码展示这两者之间的差异。首先,我们需要创建使用了关联数据片段服务器 Lua 脚本的 add_get_triple 脚本。该脚本用来创建并保存元素的 SHA1 摘要到 Redis 实例中。基于在将三元组提取到 Redis 实例时传入脚本的选项,关联数据服务器的三元组表示可能会是单一字符串、三个哈希或者三个集合,会根据策略的不同而不同。为了测试,我们将采用集合策略,为三元组中主语、谓语和宾语分别创建集合。

三元组由以下三个元素组成,由 `http://dp.la/api/items/971d73fd2dfd6376f258dd6a533697a5#sourceResource` 的 SHA1 哈希 `440bd91cb8b77850e5ca4cc648adaa3769b5a7ab` 所表示的主语,由 `http://purl.org/dc/elements/1.1/description` 的 SHA1 哈希 `2298d2f28daae5f02504a5b048d0c036ea73e3bd` 所表示的谓语,同时也是都柏林核心描述元素 (Dublin Core Description element),以及由字符串字面值的 SHA1 哈希 `fe8fb512487c2a945f03e0efcfe8c670ea861f60` 所表示的宾语。我们将使用 redis-cli 会话探索这些 Redis 键和数据结构,以便对我们想要的最终数据有个总体印象:

```
127.0.0.1:6379> GET 440bd91cb8b77850e5ca4cc648adaa3769b5a7ab
"http://dp.la/api/items/971d73fd2dfd6376f258dd6a533697a5#sourceResource"
127.0.0.1:6379> GET 2298d2f28daae5f02504a5b048d0c036ea73e3bd
"http://purl.org/dc/elements/1.1/description"
127.0.0.1:6379> GET fe8fb512487c2a945f03e0efcfe8c670ea861f60
"Relief shown by hachures. Includes 1 inset map in 2 sections: Mission du
Docteur Bayol a Timbo en 1881."
```

为了表示该三元组,我们将使用三个集合:

- 440bd91cb8b77850e5ca4cc648adaa3769b5a7ab:pred-obj 用于所有主语
的谓语和宾语
- 2298d2f28daae5f02504a5b048d0c036ea73e3bd:subj-obj 用于所有谓语的
主语和宾语
- fe8fb512487c2a945f03e0efcfe8c670ea861f60:subj-pred 用于所有宾语的
主语和谓语

我们可以使用 SCARD 命令获取每个集合的大小:

```
127.0.0.1:6379> SCARD 440bd91cb8b77850e5ca4cc648adaa3769b5a7ab:pred-obj
(integer) 25
127.0.0.1:6379> SCARD 2298d2f28daae5f02504a5b048d0c036ea73e3bd:subj-obj
(integer) 62473
127.0.0.1:6379> SCARD fe8fb512487c2a945f03e0efcfe8c670ea861f60:subj-pred
(integer) 1
```

从概念上讲, 这些集合的大小都在合理范围之内, 用于数字元素的主语图大小为 25, 普通都柏林核心描述元数据元素大小为 62437, 最后字面字符串宾语只有一个主语和谓语。每个集合中存储的是对应三元组片段的 SHA1 哈希摘要, 可以使用命令 SMEMBERS 或 SSCAN 获取这些值; 我们还可以使用命令 SISMEMBER 测试剩余的元素是否存在于集合中。我们先从主语开始:

```
127.0.0.1:6379> SSCAN 440bd91cb8b77850e5ca4cc648adaa3769b5a7ab:pred-obj 0
1) "28"
2) 1) "490fd4e5ac1e267bbd873e6df9f79f6e21bb39a4:a7efd7cea070322c1f9e0b34
641afdea7e9eed09"
2) "d82a0c98602164078281e22c3d2096c214887d4e:5972d70a66a4f74a915ffd29
a6afe09c89353986"
3) "490fd4e5ac1e267bbd873e6df9f79f6e21bb39a4:601230576f399ca97b523372
1ce327e0bbd4f206"
4) "490fd4e5ac1e267bbd873e6df9f79f6e21bb39a4:5b15ed3dd0b60e11d575d29c
640cea554cf91d82"
5) "2298d2f28daae5f02504a5b048d0c036ea73e3bd:fe8fb512487c2a945f03e0ef
cfe8c670ea861f60"
6) "54aed229df691a2f772c19d4396852b26f960827:1586bf7cee37fa79f91d94d2
210591205365432a"
7) "2298d2f28daae5f02504a5b048d0c036ea73e3bd:6a645f0ef32ac10c04201498
```



```

242bd6fa7bf7dc91"
8) "d9e34bac9b6f5b13a338df39bf61ec1965bab39d:15073464d418bafe273534b3
30ebf3a4cba5cae1"
9) "490fd4e5ac1e267bbd873e6df9f79f6e21bb39a4:c4aae22fdbf6c4799741377f
4054d2efcaa247a3"
10) "2298d2f28daae5f02504a5b048d0c036ea73e3bd:68b04fe437611ca606c29aa9
e96a7e3f8f31d707"
127.0.0.1:6379> SISMEMBER 440bd91cb8b77850e5ca4cc648adaa3769b5a7ab:predobj
2298d2f28daae5f02504a5b048d0c036ea73e3bd:fe8fb512487c2a945f03e0efcfe8
c670ea861f60
(integer) 1

```

下一步，我们将使用命令从宾语集中的众多元素中获取部分元素。我们使用 SSCAN 命令时加上了 MATCH 参数，将结果控制在我们感兴趣的匹配主语 SHA1 的那些元素：

```

127.0.0.1:6379> SSCAN 2298d2f28daae5f02504a5b048d0c036ea73e3bd:subj-obj 0
MATCH 440bd91cb8b77850e5ca4cc648adaa3769b5a7ab:* COUNT 100000
1) "0"
2) 1) "440bd91cb8b77850e5ca4cc648adaa3769b5a7ab:68b04fe437611ca606c29aa9e
96a7e3f8f31d707"
2) "440bd91cb8b77850e5ca4cc648adaa3769b5a7ab:6a645f0ef32ac10c042014982
42bd6fa7bf7dc91"
3) "440bd91cb8b77850e5ca4cc648adaa3769b5a7ab:fe8fb512487c2a945f03e0efc
fe8c670ea861f60"
4) "440bd91cb8b77850e5ca4cc648adaa3769b5a7ab:7581c0c2d4d7e33db91e689dc
a8b84e1f449ac24"
5) "440bd91cb8b77850e5ca4cc648adaa3769b5a7ab:19bc513b98fa411e352f7f10d
4ee59c1c49c036c"
127.0.0.1:6379> SISMEMBER 2298d2f28daae5f02504a5b048d0c036ea73e3bd:subjobj
440bd91cb8b77850e5ca4cc648adaa3769b5a7ab:fe8fb512487c2a945f03e0efcfe8
c670ea861f60
(integer) 1

```

我们将使用 SMEMBERS 获取宾语的所有主语和谓语：

```

127.0.0.1:6379> SMEMBERS fe8fb512487c2a945f03e0efcfe8c670ea861f60:subjpred
1) "440bd91cb8b77850e5ca4cc648adaa3769b5a7ab:2298d2f28daae5f02504a5b048d0
c036ea73e3bd"

```

我们采用了伊利诺伊大学的数据集（来源于 Dp.la 网站）作为原始 RDF 图输入，请参考附录来源第 10 章：信息流的测量与管理中的第二个要点。我们使用 Python 模块 rdflib 将 JSON 关联数据（JSON-LD）转换成 RDF 图。新的 RDF 图有如下特征：

新的 JSON-LD 文件的大小	101 MB
三元组的总数	605,150
唯一主语	132,671
唯一谓语	34
唯一宾语	227,302

我们在四核 16GB 内存的工作站上使用关联数据片段服务器上的 Lua 脚本 `add_get_triple` 获取平均 5000 个三元组数据需要花费大概 2.53 分钟，因此使用该 Python 函数需要耗费大约 5 个小时来完成所有的数据提取工作：

```
def ingest_graph(graph, lua_script_digest, datastore):
    start = datetime.datetime.now()
    counter = 0
    print("Started ingesting {} triples at {}".format(len(graph),
start.isoformat()))
    for subject, predicate, object_ in graph:
        datastore.evalsha(lua_script_digest, 3, subject, predicate,
object_)
        if not counter%10 and counter > 0:
            print(".", end="")
        if not counter%100:
            print(counter, end="")
        if not counter%100 and counter > 0:
            print(":{0} mins".format((datetime.datetime.now()-start).
seconds / 60.0))
            counter += 1
        end = datetime.datetime.now()
        print("Finished at {} total time {}".format(end, (end-start).
seconds / 60.0))
```

该数据集的 Redis 文件 `dump.rdb` 大约 181MB，总共 605,150 个 Redis 键。

如果你已经将数据抽取出来并加载至 Redis 的话，可以使用来自 <https://github.com/sripathikrishnan/redis-rdb-tools> 项目的 `rdb` 工具，将 Redis 的 `dump.rdb`

文件(或者其他任何 Redis rdb 文件)转换为 RESP 格式,然后使用块提取过程加载至 Redis。我们可以运行 `pip install rdbtools` 来安装,也可以将项目克隆下来然后运行 Python 的 `setup.py` 来安装。之后,我们就可以使用 `rdb` 可执行文件了。执行下列命令将生成 RESP 文件,并将结果导入 `dpa1_resp.tx` 文件:

```
$ rdb --command protocol dump.rdb > dpa1_resp.txt
```

之后,我们可以使用 UNIX 的 `time` 命令估算将大量数据从文件 `dpa1_resp.txt` 插入到空 Redis 实例中所需要的时间,这当中使用到了 UNIX 实用程序 `netcat`:

```
$ time (cat dpa1_resp.txt; sleep 10) | nc localhost 6379 > /dev/null
```

```
real    0m14.265s
user    0m0.127s
sys     0m1.459s
```

我们在另一个终端窗口上,获取总数之后清空 Redis 实例:

```
127.0.0.1:6379> DBSIZE
(integer) 605150
127.0.0.1:6379> FLUSHALL
OK
(1.71s)
```

最后,我们使用 `redis-cli` 程序在流水线模式下进行第二个测试:

```
$ time cat dpa1_resp.txt | ~/redis/src/redis-cli --pipe
All data transferred. Waiting for the last reply...
Last reply received from server.
errors: 0, replies: 2025798
```

```
real    0m3.750s
user    0m0.259s
sys     0m0.411s
```

现在回到 `redis-cli` 会话中,我们通过再次运行 `DBSIZE` 命令来确认数据已经加载完毕:

```
127.0.0.1:6379> DBSIZE
(integer) 605150
```

相比最初的 Python 脚本获取原始 JSON-LD 文件来说, 采用 Netcat 和 redis-cli 来提取 RESP 文件极大地减少了时间上的花费。你需要对应用程序中的数据种类与信息流多加考虑, 特别是那些用来执行常见 ETL 工作流的应用程序, 例如将大量的数据提取到 Redis 数据存储中。

在了解了这些 JSON 数据结构之后, 我们可以创建一个函数将每个主语、谓语和宾语的 SHA1 输出到 RESP 文件, 来代替使用 Redis 客户端直接向运行中的 Redis 实例发送命令。批量上传会简单一点并且绝对要快几个数量级, 即使我们需要做一些操作和 JSON 过滤, 以便将 JSON 关联数据转化成 RESP 格式, 再将数据集加载到 Redis 中。为了将 JSON 关联数据转换成 RESP, 我们需要一种方法将 RDF 记录的结构元数据过滤, 同时在计算谓语或者宾语元素的 SHA1 哈希摘要之前将个别字段扩展为完整的 URL。

将 JSON 转换成 RESP

像许多 ETL 工作流程一样, 在抽取和转换数据之前, 我们需要记录额外结构信息将数据转换成可以加载的格式。幸运的是, dp.la 提供了记录格式的详尽描述, 称为元数据应用程序配置文件, 可以在 <http://dp.la/info/developers/map/> 上找到。dp.la MAP 使用都柏林核心元数据标准 (Dublin Core metadata standard) 和欧洲数据模型 (EDM) 的组合, 每条记录都有自己的 JSON-LD 格式的命名空间。

这段代码性能堪忧。一种用来加速提取数据到关联数据片段服务器的替代方案是解析 JSON 关联数据文件, 计算主语、谓语和宾语的 SHA1 哈希, 然后生成 RESP 文本文件。我们将使用函数 `generate_redis_protocol` 输出 RESP:

```
def generate_redis_protocol(cmd):
    proto = ""
    proto += "*" + str(len(cmd)) + "\r\n"
    for arg in cmd:
        proto += "$" + str(len(arg)) + "\r\n"
        proto += arg + "\r\n"
    return proto
```

在使用函数 `generate_redis_protocol` 前, 我们首先使用导入的标准 JSON Python 模块将文件对象加载为 Python 列表, 其中的每条记录为 Python 字典类型, 然后再加载原始 JSONLD 文件:

```
>>> import json
>>> dpla_ui = json.load(
    open("/tmp/university_of_illinois_at_urbana-champaign"))
>>> len(dpla_ui)
18103
```

我们将随机展示一条 JSON 字典中的记录。这条记录将展示其结构信息，并对这条从伊利诺伊大学的数字资源库中采集的 DPLA 记录的下列缩写注释的可用信息有个大致了解。

```
>>> record = dpla_ui[5678]
```

我们来看这条示例记录，首先感受一下 JSON-LD 对象的结构和内容来确定我们想要抽取的主语、谓语和宾语，因此我们可以构建自己的抽取和转换脚本。先让我们看看这个 Python 字典顶层键的组成。

```
>>> record.keys()
dict_keys(['ingestionSequence', '_id', 'ingestType', 'admin', 'object',
'isShownAt', 'aggregatedCHO', '_rev', 'dataProvider', 'originalRecord',
'sourceResource', 'id', 'ingestDate', '@type', '@context', '@id',
'provider'])
```

JSON-LD 记录中的 @context 部分定义了所有属性的默认命名空间：

```
>>> record.get('@context')
'http://dp.la/api/items/context'
```

将 JSON 转换为 RESP 的脚本 dpla2resp.py 的部分输出需要基于 `http://dp.la/api/items/context` 和 `http://www.europeana.eu/schemas/edm/` 的 URL 来扩展所有的属性：

```
>>> record.get('@id')
'http://dp.la/api/items/b85d182ccb7d800c8c13b65743ef9ac7'
```

该记录字典中的 @id 键对应 URI `http://dp.la/api/items/b85d182ccb7d800c8c13b65743ef9ac` 的主语，我们将它设置为 SHA1 键 `d7210e2eca59b8c25086dbef406b2a41720f079c` 的值：

```
>>> item_hash = hashlib.sha1(record.get('@id').encode())
>>> item_hash.hexdigest()
'd7210e2eca59b8c25086dbef406b2a41720f079c'
```

我们将使用 `item_hash` 和 `@id` 生成我们的第一个 RESP，调用 `generate_redis_protocol` 函数，传入的列表包含 3 项内容：

```
>>> resp = generate_redis_protocol(["SETNX", item_hash.hexdigest(),
record.get('@id')])
>>> print(resp.encode())
*3\r\n
$5\r\n
SETNX\r\n
$40\r\n
d7210e2eca59b8c25086dbef406b2a41720f079c\r\n
$55\r\n
http://dp.la/api/items/b85d182ccb7d800c8c13b65743ef9ac7\r\n'
```

再回到我们一开始的 Redis 实例 `dpla_redis`，其中有已加载的伊利诺伊大学的 DPLA 三元组库。现在我们可以获取记录的所有谓语和宾语，来看看需要从 JSON 记录中抽取什么字段来创建 RESP 文件：

```
>>> dpla_redis = redis.StrictRedis()
>>> for row in dpla_redis.smembers("d7210e2eca59b8c25086dbef406b2a41720f0
79c:pred-obj"):
    print(dpla_redis.get(row.decode().split(":")[0]))

b'http://www.europeana.eu/schemas/edm/isShownAt'
b'http://www.europeana.eu/schemas/edm/provider'
b'http://www.europeana.eu/schemas/edm/dataProvider'
b'http://www.europeana.eu/schemas/edm/object'
b'http://purl.org/dc/elements/1.1/_rev'
b'http://www.europeana.eu/schemas/edm/aggregatedCHO'
b'http://www.w3.org/1999/02/22-rdf-syntax-ns#type'
b'http://dp.la/terms/SourceResource'
```

记录中的 RDF `@type` 需要扩展成完整的 URL `http://www.openarchives.org/ore/terms/Aggregation`，存储在以下命名空间的 JSON 记录中：

```
>>> record.get('@type')
'ore:Aggregation'
```

对于该主语来说，我们将从该记录字典的其他字段抽取谓语和宾语的值，包括 JSON

转换为 RESP 脚本的 `aggregation2resp` 函数中的 `isShownAt` `dataProvider`、`object`、`aggregateCHO`、`SourceResource` 和 `_rev` 键。`aggregation2resp` 函数首先获取该聚合记录的 RDF ID，为该记录的 IRI 创建 SHA1 哈希，使用 Redis 的 `MSETNX` 命令为 RDF 类 IRI 和对应的值创建新的记录。然后创建一个 Redis 集合键用来存储所有的谓词和宾语以便之后使用，其中集合键的值以 RDF 类的值作为开头：

```
def aggregation2resp(record):
    raw_protocol = ''
    record_iri = record.pop("@id")
    record_hash = hashlib.shal(record_iri.encode())
    record_type = record.pop("@type")
    rdf_type_hash = hashlib.shal(str(rdflib.RDF.type).encode())
    record_type_hash = hashlib.shal(record_type.encode())
    raw_protocol += generate_redis_protocol(
        ["MSETNX",
         record_hash.hexdigest(),
         record_iri,
         rdf_type_hash,
         str(rdflib.RDF.type),
         rdf_type_value_hash.hexdigest(),
         record_type])
    record_pred_obj = "{}:pred-obj".format(record_hash)
    raw_protocol += generate_redis_protocol(
        ["SADD",
         record_pred_obj,
         "{}:{}".format(rdf_type_hash, record_type_hash)])
```

对于该记录的键来说，`aggregation2resp` 函数会查询欧洲词汇的 `isShownAt`、`dataProvider`、`object`、`aggregateCHO` 属性。如果属性不存在于 RESP 中的话，我们会使用 Redis 的 `MSETNX` 命令添加每个属性的完整 IRI 和对应的 SHA1 哈希。

```
for key in record.keys():
    # EDM simple triples
    if key in ['isShownAt',
              'dataProvider',
              'aggregatedCHO',
              'object']:
```

```

key_iri = getattr(EDM, key)
key_hash = hashlib.sha1(key_iri.encode())
key_value = record.get(key)
key_value_hash = hashlib.sha1(key_value.encode())
raw_protocol += generate_redis_protocol(
    ["MSETNX",
     key_hash.hexdigest(),
     key_iri,
     key_value_hash.hexdigest(),
     key_value])

```

将这些 EDM 属性添加到 RESP 字符串之后, 这些来自单独的 EDM 属性的主语、谓语和宾语的对应 Redis 集合 RESP 将产生分别用于 subj_pred_key、edm_subj_obj 和 edm_subj_predReds 的键:

```

raw_protocol += generate_redis_protocol(
    ["SADD",
     record_pred_key.hexdigest(),
     "{}:{}".format(
         key_hash.hexdigest(),
         key_value_hash.hexdigest())])
edm_subj_obj = "{}:subj-obj".format(key_hash.hexdigest())
raw_protocol += generate_redis_protocol(
    ["SADD",
     edm_subj_obj,
     "{}:{}".format(record_hash.hexdigest(),
                     key_value_hash.hexdigest())])
edm_subj_pred = "{}:subj-pred".format(
    key_value_hash.hexdigest())
raw_protocol += generate_redis_protocol(
    ["SADD",
     edm_subj_pred,
     "{}:{}".format(record_hash, key_hash)])

```

aggregation2resp 函数将创建并添加 Dublin-core_rev 值到协议中:


```

if '_rev' in record:
    dc_rev_hash = hashlib.sh1(getattr(DC, '_rev').encode())
    dc_rev_value = record.get('_rev')
    dc_rev_value_hash = hashlib.sh1(dc_rev_hash.encode())
    raw_protocol += generate_redis_protocol(
        ["MSETNX",
         dc_rev_hash.hexdigest(),
         getattr(DC, '_rev'),
         dc_rev_value_hash.hexdigest(),
         dc_rev_value])
    raw_protocol += generate_redis_protocol(
        ["SADD",
         record_pred_key,
         "{}:{}".format(dc_rev_hash.hexdigest(),
                        dc_rev_value_hash.hexdigest())])
    raw_protocol += generate_redis_protocol(
        ["SADD",
         "{}:subj-obj".format(dc_rev_hash.hexdigest()),
         "{}:{}".format(record_hash.hexdigest(),
                        dc_rev_value_hash.hexdigest())])
    raw_protocol += generate_redis_protocol(
        ["SADD",
         "{}:subj-pred".format(dc_rev_value_hash.hexdigest()),
         "{}:{}".format(record_hash.hexdigest(),
                        dc_rev_hash.hexdigest())])

```

对于聚合记录的 `sourceResource`、`provider` 和 `originalRecord` 谓语来说, 这些值其他 RDF 主语也需要由 `dpla2resp.py` 脚本产生的 RESP 输出。这三个 RESP 生成函数分别为 `sourceResource2resp`、`provider2resp` 和 `originalRecord2resp`。对于我们的示例记录来说, 我们可以检查每个 Python 字典并为聚合记录属性生成 SHA1 哈希。在这里, 我们可以从 Python shell 中看到 `sourceResource` 属性:

```

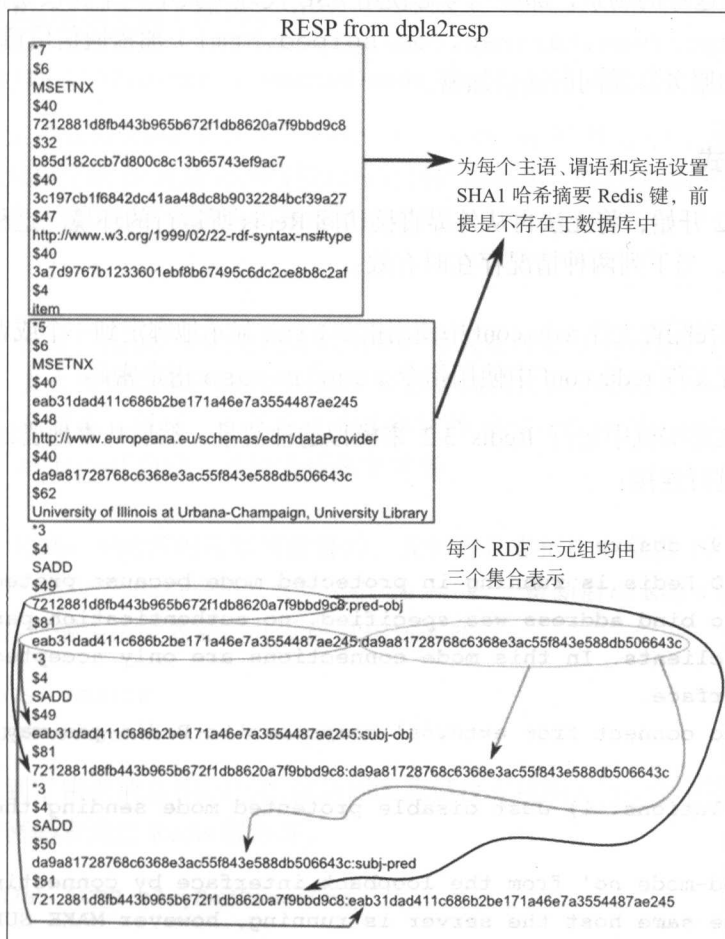
>>> source_resource_subject = record.get('sourceResource').get('@id')
>>> source_resource_subject_hash = hashlib.sh1(source_resource_subject.
encode())
>>> print(source_resource_subject, source_resource_subject_hash.
hexdigest())

```

`http://dp.la/api/items/b85d182ccb7d800c8c13b65743ef9ac7#sourceResource`
`e777561816eb6ff634b310cbb3e19ad09cdd2a4c`

在 `dp1a2resp` 函数中,这些主语的其他函数被扩展,同时 RESP 被生成并添加到 RESP 流中,然后被传入到块上传过程中。

到现在为止,我们看到的 Python 代码太过详细了,是时候重构来简化并推广一些常见的模式,这个留给读者自行探索。`aggregation2resp` 函数的输出 RESP 展现在下图中:



管理 Redis 时的安全考虑

对于 Redis 的批评在于其较弱的开箱即用的安全性或缺少安全防护。在 Redis 3.2 以前,

在公共服务器上启动 Redis 实例会将服务器暴露给连接到默认端口 6379 上的客户端。有关 Redis 安全的主要运营假设是 Redis 运行在一个安全的网络环境中, 只有可信的客户端访问数据库。像实现了缓存的 Web 服务器这样的应用程序在访问 Redis 时会提供必要的安全措施, 以便将 Redis 从用户的直接交互产生非信任输入中隔离开。

有关 Redis 安全的通用建议是使用其他技术, 至少应该采用防火墙。如果运行的是公共服务器, 则需要对 Redis 实例与外界之间采取访问控制。如果是客户端需要从公网连接到 Redis 服务器这样的场景, 那应当考虑使用 SSL 代理 (其中一个建议是使用 spiped, 详情请参考 <http://www.tarsnap.com/spiped.html>) 加密通信信道。Redis 本身并不提供客户端和服务端间的通信加密。

Redis 安全模式

从 Redis 3.2 开始, 将无法再从外界直接访问 Redis 所运行的环境。这种运营模式叫作 Redis 安全模式, 当下列两种情况存在时有效:

- 服务器没有在配置文件 `redis.conf` 中使用指令 `bind` 显示地绑定到一个或者多个 IP 地址上
- 没有在配置文件 `redis.conf` 中使用指令 `requirepass` 指定密码

我们可以在虚拟机中运行 Redis 3.2 来模拟这种情景, 然后从本地尝试使用 Redis 的 Python 客户端进行连接:

```
127.0.0.1:6379> dbsize
```

```
(error) DENIED Redis is running in protected mode because protected mode is enabled, no bind address was specified, no authentication password is requested to clients. In this mode connections are only accepted from the loopback interface.
```

```
If you want to connect from external computers to Redis you may adopt one of the
```

```
following solutions: 1) Just disable protected mode sending the command 'CONFIG
```

```
SET protected-mode no' from the loopback interface by connecting to Redis from the same host the server is running, however MAKE SURE Redis is not publicly accessible from internet if you do so. Use CONFIG REWRITE to make this change permanent.
```

```
2) Alternatively you can just disable the protected mode by editing the Redis configuration file, and setting the protected mode option to 'no',
```

and then restarting the server.

3) If you started the server manually just for testing, restart it with the '--protected-mode no' option. 4) Setup a bind address or an authentication password. NOTE: You only need to do one of the above things in order for the server to start accepting connections from the outside.

你可以从这个错误信息中看到,为了禁用 Redis 保护模式,需要在同一环境中使用 Redis 客户端连接到 Redis 服务器上使用 CONFIG SET 命令:

```
127.0.0.1:6379> CONFIG SET protected-mode no
```

你可以将文件 redis.conf 中的指令 protected-mode 修改为 no, 使用指令 bind 将起到限制只能从限定的 IP 地址和网络接口进行访问的作用, 或者使用指令 requirepass 为客户端设置在连接 Redis 实例时需要提供的密码。



注意! Redis 中的密码是以明文的方式进行存储和传播的! 如果攻击者能够在客户端与 Redis 服务器端间的流量上进行窥探, 密码就会泄露。同样, 如果攻击者可以访问 Redis 服务器上的配置文件 redis.conf 的话, 密码同样也会泄露。

为了理解 Redis 中的密码是如何设置的, 我们将在虚拟机中重启 Redis 服务器, 并在 Redis 配置文件中添加简单的密码 v3ryBadPassw0rd。重新启动 Redis 服务实例, 我们将使用 redis-cli 客户端连接服务器, 并发送 DBSIZE 命令:

```
127.0.0.1:6379> DBSIZE
(error) NOAUTH Authentication required.
```

客户端收到了错误消息 NOAUTH, 这是因为我们没有使用 AUTH 命令将定义在 redis.conf 配置文件中的密码发送给 Redis 服务器:

```
127.0.0.1:6379> AUTH v3ryBadPassw0rd
OK
127.0.0.1:6379> DBSIZE
(integer) 0
```

在使用 AUTH 命令和正确的密码认证成功之后, 接下来客户端发起的请求都能被 Redis

服务器正常处理。在主从复制架构中，如果主 Redis 实例设置了密码，那么主节点所对应的从节点需要设置其 `redis.conf` 文件中的 `masterauth` 配置指令。

命令混淆

虽然 Redis 没有类似针对特定命令的访问控制来满足综合环境下客户端可能需要减少权限的情况，但是 Redis 提供了禁用或者混淆 Redis 命令的方法。如果客户端受到威胁，某些诸如 `CONFIG SET`、`FLUSHALL` 或者任何写命令，对于连接的客户端来说都将变得不可用或者“伪装”起来。

为了演示如何使用 `rename-command` 命令进行命令混淆，我们将复制一份 `redis.conf` 并添加下列行：

```
rename-command CONFIG aReallyLongConfigtoGuess
rename-command FLUSHALL ""
rename-command DEL ""
```

将 Redis 命令设置为空的字符串 “” 会禁用该命令。我们将启动新的 Redis 3.2 服务器进行测试，使用 `redis.conf` 配置文件，然后使用 `redis-cli` 会话尝试以下命令：

```
127.0.0.1:6379> CONFIG GET maxmemory
(error) ERR unknown command 'CONFIG'
127.0.0.1:6379> aReallyLongConfigtoGuess GET maxmemory
1) "maxmemory"
2) "0"
127.0.0.1:6379> FLUSHALL
(error) ERR unknown command 'FLUSHALL'
```

在重命名 Redis 命令时，确保不要让命令以数字开头，否则 Redis 将进入死循环中！其他你可能想要重命名的命令包括 `KEYS`、`PEXPIRE`、`DEL`、`SHUTDOWN`、`BGREWRITEAOF`、`BGSAVE`、`SAVE`、`SPOP`、`SREM`、`RENAME` 和 `DEBUG`。

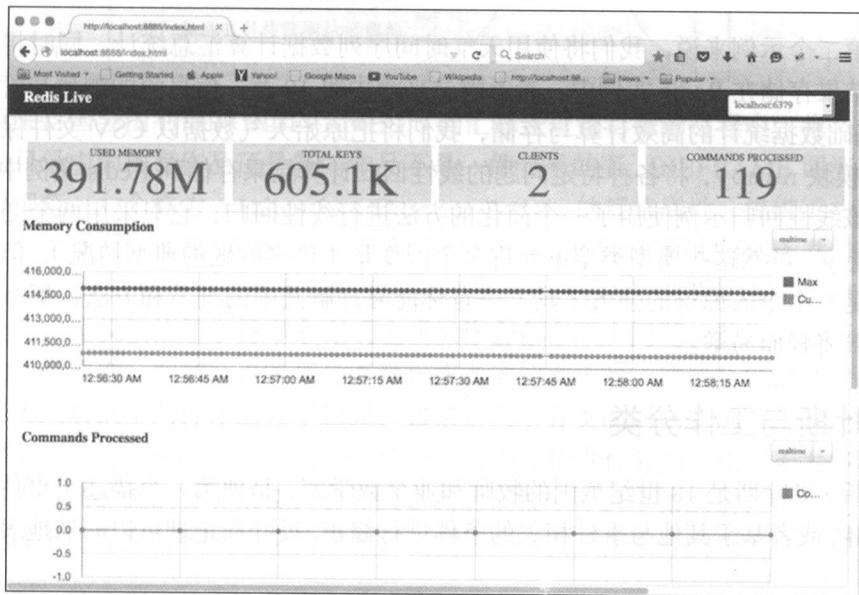
使用 Redis Web 仪表板进行运营监测

使用基于 Web 的运营仪表板监测多个系统的状态是很常见的。有一些开源的仪表板项目允许实时观测运行中的 Redis 实例，包括主从节点或者 Redis 集群的复杂拓扑结构。商业 Redis 托管公司，例如 Redis Labs，提供类似的服务，通过仪表板接口监控和应对问题。Nitin Kumar 于 2012 年撰写了一篇博客，请参考附录中来源的第 10 章：信息流的测量与管理第

三要点，宣告了 RedisLive 的发布。这款仪表板主要使用 Redis 的 INFO 命令和 MONITOR 命令监测 Redis 实例。

你可以从 GitHub 上的 <https://github.com/nkrode/RedisLive> 找到 RedisLive。它提供了基于 Web 的仪表板，用于展示受监测的 Redis 实例的活动及当前延迟峰值。RedisLive 需要至少一个专门的 Redis 实例存储统计数据，不过也可以选择使用 SQLite 数据库来代替。

除了像 RedisLive 监控 Redis 这样的基于 Web 的专用监控工具外，用于系统监测的 Android 和 iOS 这样专门的原生应用，为监测系统提供了更多的方法。



使用 RedisLive 仪表板监测 Redis

机器学习

一方面“大数据”的炒作周期将一直持续下去，另一方面 Redis 提供了多种方式确实实现了广告和媒体许诺给商业用户和领导的功能。除了作为快速加载和操作数据的明智选择之外，Redis 也为处于过渡模式中的数据搭建了分阶段平台，依据应用程序的不同，这些数据之后将被操作导向最终状态。在机器学习技术方面，Redis 扮演了数据存储的角色，对

于特别的学习算法来说，Redis 非常灵活。

本节将介绍两种监督学习任务，即朴素贝叶斯和线性回归，用来演示不同的统计分析方法。我们会将 Redis 用作中间结果的临时存储。第一个例子，数据集中包含的是之前就存在的 52 条 MARC21 记录，有关简奥斯汀的《傲慢与偏见》和赫尔曼·梅尔维尔的《白鲸》。该数据集合将被转换为 BIBFRAME 实体，然后随机分解为两个数据集合，其中一个用来训练算法，另一个用来测试朴素贝叶斯数据集合。对于线性回归示例来说，我们将使用三份公开的数据集合，并将它们组合为主数据集，之后我们会再次将其拆分为两个随机的组，就像我们对朴素贝叶斯的示例那样，其中一个用来测试线性回归而另一个用来测试结果。

对于第二个示例来说，我们将使用天气时间序列数据计算汇总统计，同时将均值和方差等计算结果存储在 Redis 实例中。参考附录来源中第 10 章：信息流的测量与管理，使用 Redis 对基础数据统计的高效计算与存储，我们将把原始天气数据以 CSV 文件保存，并使用 Python 模块 `numpy`，将各种特定问题的线性回归计算结果存储到 Redis 实例中。为了便于演示，该线性回归示例使用了一个简化的方法进行线性回归，它只采用两个变量自变量 x 和应变量 y 。虽然这些模型类型不允许多个因变量（许多问题的典型情况），但是这个简化的模型是一个令人惊讶的强大工具。一旦你能够理解其中的细节和方法，那么计算多变量模型回归将轻而易举。

朴素贝叶斯与工作分类

托马斯·贝叶斯是 18 世纪英国的牧师和业余数学家，是他第一个描述了事件的可能性该如何计算，或者基于其他与事件相关的条件进行修正。贝叶斯定理可以简洁地表达如下：

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

在这个公式中，事件 B 发生时事件 A 的概率是通过计算 A 发生时 B 的概率同 A 的概率的乘积，并除以事件 B 的概率，事件的**条件概率**依赖于之前存在的事件。朴素贝叶斯通常用于分类任务，识别垃圾邮件或者网站中评论区的垃圾信息。

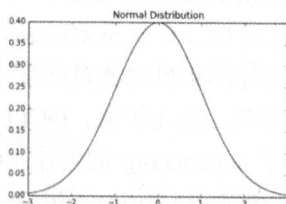
朴素贝叶斯 vs. 标准概率

$$\text{后验概率} = \frac{\text{先验概率} \times \text{可能性函数}}{\text{证据}}$$

贝叶斯概率是一个数量，被赋予了基于当时知识的假设。朴素贝叶斯是一种简单的形式，它强制假设各特征之间的无关性。

关键差异：在测试之前，朴素贝叶斯中的假设被分配了一个前提，而在标准概率上则没有。

标准概率或者称之为频率概率，将事件可能性定义为大量实验中相对频率的限制。每次实验假定是随机和独立的。事件发生的相对频率是指该事件的可能性并且往往服从经典的钟型曲线。



在使用来自国会图书馆的 BIBFRAME 1.0 词汇开发可用目录的过程中，将一件未知的作品分类为目录中的现存作品还是新的作品不是一件容易的事。BIBFRAME 词汇表构建了自有的资源规范，一个称为 `authorizedAccessPoint` 的规格属性，提供了一种粗糙的用来去除重复实体的机制。该机制的作用方式为如果两件作品共享同一个 `authorizedAccessPoint`，那么它们就被视为是同一件作品。该方法对于快速分析来说是可以接受的，但仅仅依靠字符串来完成这项工作会存在不少问题。首先，即便是标题的原始记录的细微差别，例如标点符号是否存在或者词的大小写，这些会导致经过自动化处理的严格分类之后认定两件作品并不相同，如果是人工分类的话则会认为两者是一样的。只对作品的 `authorizedAccessPoint` 使用字符串比较进行分类，相对于大多数图书管理员和编目人员所熟悉的分类来说，会导致更高的误报率（这里指的是，作品应当被分类为一致但实际上却没有），同时也会为顾客和终端用户在尝试搜索分类查找材料时带来麻烦。

我们需要一种更宽容的方法负责处理 `authorizedAccessPoint` 中的这些细微差异，同时在将潜在作品识别为先前存在于分类中的作品时又能强大到足以最小化误判率。另一个复杂的因素是在 2015 年早些时候，国会图书馆发布了 BIBFRAME 的 2.0 版本，从大多数类别中移除了 `authorizedAccessPoint` 属性，这使得开发一个用于去除重复的算法变得更关键、更复杂。幸运的是，我们可以使用朴素贝叶斯来训练算法，使用多种输入而非仅仅依赖于针对废弃的 `authorizedAccessPoint` 元素上的字符串匹配方式。

在朴素贝叶斯用来去除 BIBFRAME 中重复作品的第一轮迭代中，我们使用作品中经常出现的两种特征，即作品的标题和作品的作者，用来判定新作品是否能够分类为之前作品的可能性。理想情况下，该方法会负责处理标题和作者中细小的拼写变化以最小化错误识

别，这些错误识别的原因在于算法错误地将作品识别为先前存在的作品，不过更有可能的是这些细微的变化会导致真正相同的两件作品被区别对待。作品识别中的另一个复杂因素在于书目理论中对于作品组成的定义经不起考验或者一致的定论。实践中这种划分通常更实用主义，并且是数据驱动的。

作为图书馆服务行业中的翘楚，OCLC 在 2003 年的文章中概述了其用来聚集相似作品的方法和算法，请参考附录的来源中的第 10 章：信息流的量化与管理中的第 5 要点。在这篇文章中，WorldCat（它是 OCLC 最重要的联合目录名称，主要来自北美图书馆资料成千上百万的记录组成）中的作品集群是通过匹配主要条目（典型情况是作品的作者）和从标题中提取的关键词进行识别的。这几年来，OCLC 尝试了类似的方法，并于 2014 年宣告所有 WorldCat 记录将提供基于 schema.org 的创作（CreativeWork）定义进行作品识别，并在网站 <http://www.oclc.org/developer/develop/linked-data/worldcatentities/worldcat-work-entity.en.html> 上予以说明。

创建训练和测试的数据集

我们将以两份 MARC21 `pride-and-prejudice.mrc` 和 `moby-dick.mrc` 文件开始，使用国会图书馆的 MARC2BIBFRAME Xquery 项目将这些记录处理并转换成 BIBFRAME RDF 图。该项目托管在 GitHub 上的 <https://github.com/lcnetdev/marc2bibframe>。在该国会图书馆项目中，作品是基于 MARC21 字段和记录被识别和创建的，但是没有在本地或者远程大型目录进行去重。在 Python shell 中，我们首先导入 `pymarc` Python 模块，并创建两个 MARC21 记录列表：

```
>>> import pymarc
>>> pride_and_prejudice_recs = [r for r in
pymarc.MARCReader(open("prideand-prejudice.mrc", "rb"), to_unicode=True)]
>>> moby_dick_recs = [r for r in pymarc.MARCReader(open("moby-dick.mrc",
"rb"), to_unicode=True)]
>>> len(pride_and_prejudice_recs)
30
>>> len(moby_dick_recs)
22
```

下一步，我们将导入 Redis Python 客户端，创建 Redis 实例，然后加载之前测试过的 Lua 脚本 `add_get_triple`：

```

>>> import redis
>>> bayes_datastore = redis.StrictRedis()
>>> bayes_datastore.dbsize()
0
>>> with open("linked-data-fragments/redis_lib/add_get_triple.lua") as
lua_file:
    raw_lua = lua_file.read()

>>> add_get_triple_digest = bayes_datastore.script_load(raw_lua)
>>> add_get_triple_digest
b'6ac0387e16f9408cece6502a279a4d4c8971bf97'

```

在继续之前，我们需要导入标准的 Python 模块 socket 及第三方模块 rdflib:

```
>>> import rdflib, socket
```

现在，我们将 MARC XML 文件传入通过 xquery_socket 函数连接的套接字服务器上，并取回 MARC 记录的 RDF BIBFRAME 图数据，并存储到第二个 Python 列表 all_graphs 中:

```

>>> for recs in [pride_and_prejudice_rec, moby_dick_rec]:
    for marc_record in recs:
        all_graphs.append(xquery_socket(pymarc.record_to_xml(marc_record,
namespace=True)))
>>> len(all_graphs)
52

```

为了协助将这些图数据提取到关联数据片段服务器上，我们将创建 process_graph 函数，它会将每个图数据的主语、谓词和宾语都调用 add_get_triple Lua 脚本:

```

def process_graph(graph):
    for s,p,o in graph:
        bayes_datastore.evalsha(
            add_get_triple_digest,
            3,
            str(s),
            str(p),
            str(o))

```

现在有了 BIBFRAME 图之后，我们通过对 `all_graphs` 列表中的每个图调用 `process_graph` 函数来分别处理。最后，我们检查 Redis 数据库在存入了 RDF 图之后的大小：

```
>>> for graph in all_graphs:
    process_graph(graph)
```

```
>>> bayes_datastore.dbsize()
10433
```

我们通过在 `5d1377f4476a1cbfb3caea106dc6b0a7d086410a:subj-pred` 键上执行 `SMEMBERS` 命令获取并存储所有代表 BIBFRAME 作品 IRI `http://bibframe.org/vocab/Work` 的主语-谓语。在该示例中所有谓语应当为代表 `http://www.w3.org/1999/02/22-rdf-syntax-ns#type` 的 RDF 类型 IRI 的 SHA1 哈希摘要 `3c197cb1f6842dc41aa48dc8b9032284bcf39a27`。然后，我们通过使用 Python 字符串 `split` 方法获取主语：

```
>>> bayes_datastore.scard(
'5d1377f4476a1cbfb3caea106dc6b0a7d086410a:subj-pred')
114
>>> work_digests = []
>>> for row in bayes_datastore.smembers(
'5d1377f4476a1cbfb3caea106dc6b0a7d086410a:subj-pred'):
    work_digests.append(row.decode().split(":")[0])
>>> len(work_digests)
114
```

有了这 114 份作品主题摘要之后，现在我们能够随机将这些主语分为两个 Redis 集合 `bf-training` 和 `bf-testing`，并存储于数据库中：

```
>>> for subject in work_digests:
    if random.random() >= .5:
        bayes_datastore.sadd("bf-training", subject)
    else:
        bayes_datastore.sadd("bf-testing", subject)
```

这两个集合大小上很接近，并且是我们所期望的均匀随机排序：

```
>>> bayes_datastore.scard("bf-training")
```

59

```
>>> bayes_datastore.scard("bf-testing")
```

55

既然我们已经测试并训练了数据集，那么我们将开始把朴素贝叶斯用于 BIBFRAME 作品识别之上，使用开源的 Python 库 `redisbayes`，可以从 GitHub 上的 <https://github.com/jart/redisbayes> 获取。`redisbayes` 模块需要一串单词，称为标识，是从我们感兴趣的宾语中提取出来的，用来归类是否为作品的一部分。

从 BIBFRAME 作品中提取单词标识

这些记录是使用 BIBFRAME 1.0 转换器转换而来的，我们可以为每件作品使用 `authorizedAccessPoint` 来简化示例。在把作品的 `authorizedAccessPoint` 用作测试之前，我们使用 Python shell 和 SHA1 哈希摘要 `a548a25005963f85daa1215ad90f7f1a97fbe749` 再次检测数据库中的那些拥有 `authorizedAccessPoint` 的作品，以便用于测试或者用于训练贝叶斯应用程序。

```
>>> total_missing_auth_pts = 0
>>> for i,digest in enumerate(work_digests):
    result = bayes_datastore.sscan("{}:pred-obj".format(digest),
                                   match="a548a25005963f85daa1215ad90f7f1a97fbe749:*")
    if len(result[1]) < 1:
        total_missing_auth_pts +=1
```

```
>>> total_missing_auth_pts
```

21

从结果中可以看到，有 21 条数据缺少 `authorizedAccessPoint`，这意味着 `marc2bibframe` 转换程序没有在转换过程中为所有的 BIBFRAME 作品生成该属性。取而代之的是，我们将确保所有的作品至少含有以下两者之一。

- **BIBFRAME title:** <http://bibframe.org/vocab/title> 及 SHA1 哈希摘要 `e366a989e4becead9409ca4d44ddf307afcl26b3`。
- **BIBFRAME workTitle:** <http://bibframe.org/vocab/workTitle> 及 SHA1 哈希摘要 `f610f749c5c2eaf6718eb2bc24bf74559d14637d`。这通常是一个 IRI，解析为标题 BIBFRAME 类实例的另一个资源。

在对应本章的 Python 代码文件 `bayes_works.py` 中, `generate_work_tokens` 函数接收主语摘要和关联数据片段服务器 Redis 实例 (默认地址 `http://localhost:6379`), 并创建一个标识列表和作品的谓语-宾语 Redis 键 `work_pred_objs`, 然后使用 Redis 的 `SCARD` 命令获取 `work_pred_objs` 的大小并存储到 `total_triples` 变量中:

```
def generate_work_tokens(
    work_digest,
    datastore=redis.StrictRedis()):
    tokens = []
    work_pred_objs = "{}:pred-obj".format(work_digest)
    total_triples = datastore.scard(work_pred_obj)
```

我们已知每件作品都应该含有 `title` 或者 `workTitle`, 接下来, 代码将通过使用 Redis 命令 `SSCAN` 的 `glob` 模式匹配获取 `BIBFRAME title`, 同时显示地将 `count` 设置为 `total_triples` 变量。最后, 使用得到的结果调用内部函数 `extend_tokens`:

```
bf_title_result = datastore.sscan(work_pred_objs,
    match="e366a989e4becead9409ca4d44ddf307afc126b3:*",
    count=total_triples)
if len(bf_title_result[1]) > 0:
    extend_tokens(bf_title_result[1])
```

`extends_tokens` 函数的实现是从数据库中获取 `BIBFRAME title` 的值并将其拆分为单词标识, 并在追加到 `tokens` 列表前进行小写转换:

```
def extend_tokens(result):
    for row in result:
        first_key, second_key = row.decode().split(":")
        value = datastore.get(second_key)
        tokens.extend([word.lower() for word in value.split()])
```

如果 `bf_title_result` 是空的列表, 那么代码会尝试获取代表 `workTitle` 的 `SHA1`。如果能查找到结果, 那么代码会获取 `Title` 类的 `SHA1`, 然后获取 `BIBFRAME titleValue`, 并用得到的结果调用 `extend_tokens`:

```
else:
    bf_work_title_result = datastore.sscan(work_pred_objs,
        match="f610f749c5c2eaf6718eb2bc24bf74559d14637d:*",
```

```

count=total_triples)
if len(bf_work_title_result[1]) > 0:
    for row in bf_work_title_result[1]:
        rdf_title_key = row.decode().split(":")[1]
        rdf_title_value_result = datastore.sscan(
            "{}:pred-obj".format(rdf_title_key),
            "0859add153c1fcda5e32853e22ccfe8514702b2e:*")
        if len(rdf_title_value_result[1]) > 0:
            extend_tokens(bf_work_title_result[1])

```

在使用代表 BIBFRAME 标签的 SHA1 哈希为每个创作者或者贡献者的标签返回完整的标识列表前, generate_work_tokens 函数的最后一部分尝试从所有的 BIBFRAME 创作者 (<http://bibframe.org/vocab/creator> 和哈希摘要 0f08c96e756a4fa720257bf3090efdf76b5d3acc) 和 BIBFRAME 贡献者 (<http://bibframe.org/vocab/contributor> 和哈希摘要 a20301af19937f3787275c059dae953eaff2cb5f) 中获取所有的命名信息:

```

for key_digest in ["0f08c96e756a4fa720257bf3090efdf76b5d3acc",
                  "a20301af19937f3787275c059dae953eaff2cb5f"]:
    bf_result = datastore.sscan(
        work_pred_objs,
        match="{}:*".format(key_digest),
        count=total_triples)
    if len(bf_result[1]) > 0:
        for row in bf_result[1]:
            agent_key = row.decode().split(":")[1]
            agent_scan_result = datastore.sscan(
                "{}:pred-obj".format(agent_key),
                match="56375fdb9714268c237e4eb7e74f6f0544098935:*")
            count=100)
            if len(agent_scan_result[1]) > 0:
                extend_tokens(agent_scan_result[1])
return tokens

```

应用朴素贝叶斯

现在, 我们有了一个用来产生单词标识的函数, 我们回到用于测试朴素贝叶斯公式的训练集 bf-training 上。那么举例来说, *Pride and Prejudice* 的贝叶斯定理方程如下所示:

- $p(c|x,y)$ 代表的是, 在给定 “*Pride and Prejudice*” 的标题和作者 *Jane Austen* 的前提下, 该 BIBFRAME 作品是 *Pride and Prejudice* 的概率
- $p(x,y|c)p(c)/p(x,y)$ 代表的是在给定作品是 *Pride and Prejudice* 的前提下标题是“*Pride and Prejudice*” 并且作者是 *Jane Austen* 的概率乘以 BIBFRAME 作品的概率, 所得的乘积再除以作品是 *Pride and Prejudice* 的概率

为了简化这些概率的计算, 我们假定这些特征之间是独立的。在特征独立的假设下, 标题和作者中的每个词汇在统计上是相互独立的。虽然这种朴素的假设极其可能不为真(也就是说, 词汇 *Pride* 和 *Prejudice* 同时出现在作品标题中的概率要高于各自单独出现的概率), 但是在实际操作中这种假设对于分类任务来说很有效果。

现在, 我们回到 `bayes_works.py` 模块中, `train` 函数接收作品的训练集, 为每件作品生成单词标识字符串, 然后提示用户为每件作品进行手工分配, 如果作品是 *Pride and Prejudice* 的话就分配 **pp**, 如果作品是 *Moby Dick* 的话就分配 **md**, 如果作品是未知的话就分配 **uk**。为了演示我们所说的内容, 我们将启动一个 Python 2.7 的 shell (`redisbayes` 模块当前运行在 Python 2.7 下), 导入必需的模块。同时, 我们将启动第二个 Redis 实例, 运行在 6380 端口上以供 `redisbayes` 模块使用, 使用该 Redis 实例创建 `redisbayes` 对象, 并检测两个 Redis 实例的大小:

```
>>> import redis, redisbayes
>>> ldfs = redis.StrictRedis()
>>> bayes_datastore = redis.StrictRedis(port=6380)
>>> rb = redisbayes.RedisBayes(redis=bayes_datastore)
>>> print(ldfs.dbsize(), bayes_datastore.dbsize())
(10435L, 0L)
```

我们将我们的 `bayes_works.py` 模块导入 Python shell 中, 并在 Redis 集合 `bf-training` 中创建所有的作品摘要键的列表:

```
>>> import bayes_works
>>> training_works = list(ldfs.smembers('bf-training'))
>>> len(training_works)
59
```

对于 `training_works` 列表中的每个 BIBFRAME 作品摘要来说, 代码会调用 `generate_work_tokens` 函数, 并调用 `rb.train` 方法将单词字符串进行分类和存储 (此处选择展示了部分被分类的作品, 以替代训练集中所有 59 份作品):

```
>>> for digest in training_works:
...     tokens = bayes_works.generate_work_tokens(digest)
...     print(tokens)
...     classify = input("Classify pp, md, uk> ")
...     rb.train(classify, ' '.join(tokens))
['mansfield', 'park', 'austen,', 'jane,', '1775-1817.', 'wiltshire,',
'john.']
Classify pp, md, uk> "uk"
['short', 'stories.', 'selections.', 'hawthorne,', 'nathaniel,', '1804-
1864.']
Classify pp, md, uk> "uk"
['moby', 'dick.', 'melville,', 'herman,', '1819-1891.']
Classify pp, md, uk> "md"
['pride', 'and', 'prejudice', 'austen,', 'jane,', '1775-1817.']
Classify pp, md, uk> "pp"
```

在查看了训练集并分类了所有作品到这三个分类中之后，我们将开启一个 redis-cli 会话连接上 Redis 实例。该 Redis 实例存储了 redisbayes 模块使用的用于朴素贝叶斯计算的中间数据：

```
127.0.0.1:6380> dbsize
(integer) 4
127.0.0.1:6380> keys *
1) "bayes:md"
2) "bayes:pp"
3) "bayes:uk"
3) "bayes:categories"
```

bayes:md、bayes:uk 和 bayes:pp 这三个 Redis 哈希中的每个字段都是单词标识，字段对应的值包含了词汇在训练会话中出现的总共次数。bayes:categories 包含了所有不同的分类的集合，也就是 **md**、**uk** 和 **pp**。在完成了对朴素贝叶斯实现的训练之后，我们将接收来自 bf-testing 集合的三个随机作品摘要键，然后从 Python 2.7 shell 上看看 redisbayes 实例会将这些作品分类成哪三个选项：

```
>>> test_work_1 = ldfs.srandmember("bf-testing")
>>> test_work_1
'bffcd57ebcad72b7f5a98a8cc3e7eac178815dbbb'
>>> test_work_2_tokens = bayes_works.generate_work_tokens(test_work_1)
```



```
>>> test_work_2_tokens
['pride', 'and', 'prejudice.', 'austen,', 'jane,', '1775-1817.']
>>> rb.classify(' '.join(test_work_2_tokens))
'pp'
>>> test_work_2 = ldfs.srandmember("bf-testing")
>>> test_work_2
'448910eeaa3908bab9213cff291074667872adfa'
>>> test_work_2_tokens = bayes_works.generate_work_tokens(test_work_2)
>>> test_work_2_tokens
['moby', 'dick,', 'or,', 'the', 'whale', 'melville,', 'herman,', '1819-1891.']
>>> rb.classify(' '.join(test_work_2_tokens))
'md'
```

朴素贝叶斯分类器的性能对于这个小型示例来说可以满足预期。明确地讲，即使是在小型图书馆的所有作品上应用该方法，也需要额外修改及对标记方法和处理代码的扩展。如果你需要使用简单的朴素贝叶斯机器学习方法对输入的数据流进行分类，那么希望这个示例能够在你设计 Redis 应用程序时有所帮助。

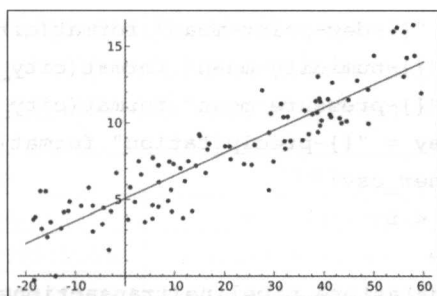
使用 Redis 实现线性规划

线性回归作为众多领域中的基础统计技术，包括医学、经济学、心理学和一般数据分析，尝试预测连续数据源的目标值。通常，使用线性回归容易对结果进行解释（也就是说，因变量是回归权重修改一到多个自变量的结果），同时，相比其他机器学习技术而言，在计算上更为轻量级。线性回归的弱点在于其性能不佳，并且当应用于非线性数据时会导致不精确的结果。

制定和编写一个线性回归解决方案需要从收集和准备数据开始。线性回归需要数值和任何离散的非数值来映射为二进制值。对于我们的测试实验来讲，我们假设标量因变量 y 和一到多个解释性自变量之间有简单的线性关系。对于我们的回归模型来说，我们将使用最小二乘拟合法，尝试将线性回归模型中所有方程的结果误差的平方和最小化。最小二乘法模型如下所示：

$$y = \alpha + \beta x$$

其中数据由具有标量响应 $y(i)$ 的 n 个观察值组成，并且其中模型中的参数，缺失的 α 字符代表截距，缺失的 β 字符代表斜率系数。



摘自 https://en.wikipedia.org/wiki/Simple_linear_regression#/media/File:Linear_regression.svg 的简单线性回归模型

为了演示如何使用简单线性回归,我们将使用来自三个不同城市的 2014 年的天气时间序列数据集。这三个城市分别是丹佛、东京和开普敦,可以从地下天气 <http://www.wunderground.com/> 网站上找到。这些数据集是采用逗号分隔的格式,列中包含了日期、温度、露点、湿度、气压、风速、降水和云层覆盖。可以从本书的网站或者 GitHub 仓库中下载 `denver-2014.csv`、`tokyo-2014.csv` 和 `nairobi-2014.csv` 这三份文件。我们将开始对天气进行分析。我们会有选择性地存储某些变量,例如平均温度 C (用摄氏度表示)、露点 C (用摄氏度表示)、平均湿度、平均海平面气压 hPa (单位帕),以及降雨量 mm (用毫米表示)。

为了将这些字段加载到 Redis 实例中,我们将创建一个 Python 函数 `extract_load`,使用标准 CSV 模块读取 CSV 文件,提取那些我们想要存入 Redis 的字段,为每个城市创建五个哈希,每个哈希对应一个变量,然后将变量作为哈希值添加进来,对应的字段名为日期字符串:

```
def extract_load(datastore=redis.StrictRedis()):
    for filename in ['cape-town-2014.csv',
                    'denver-2014.csv',
                    'tokyo-2014.csv']:

        weather_csv = csv.reader(open(os.path.join(
            CURRENT_DIR,
            filename)))

        field_names = next(weather_csv)
        city_date = filename.split(".")[0]
        temp_key = "{}-temp-mean".format(city_date)
```

```

dew_point_key = "{}-dew-point-mean".format(city_date)
humidity_key = "{}-humidity-mean".format(city_date)
pressure_key = "{}-pressure-mean".format(city_date)
precipitation_key = "{}-precipitation".format(city_date)
for row in weather_csv:
    if len(row) < 8:
        continue
    pipeline = datastore.pipeline(transaction=True)
    date_field = row[0]
    pipeline.hsetnx(temp_key, date_field, row[2])
    pipeline.hsetnx(dew_point_key, date_field, row[4])
    pipeline.hsetnx(humidity_key, date_field, row[8])
    pipeline.hsetnx(pressure_key, date_field, row[11])
    pipeline.hsetnx(precipitation_key, date_field, row[19])
    pipeline.execute()
    print("Finished {}".format(city_date))

```

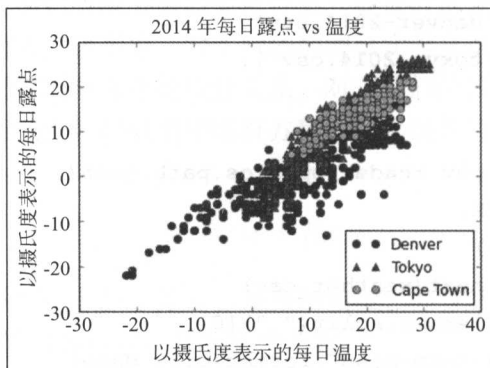
在所有 2014 年的数据加载到用于回归的 Redis 实例之后，我们会做一个快速的完整性检查，确保数据满足我们的预期：

```

127.0.0.1:6379> HLEN cape-town-2014-temp-mean
(integer) 365
127.0.0.1:6379> HLEN denver-2014-temp-mean
(integer) 365
127.0.0.1:6379> HLEN tokyo-2014-temp-mean
(integer) 365

```

由于按照定义露点是依赖于温度的，因而我们期望它们之间是正向关系，如下图所示：

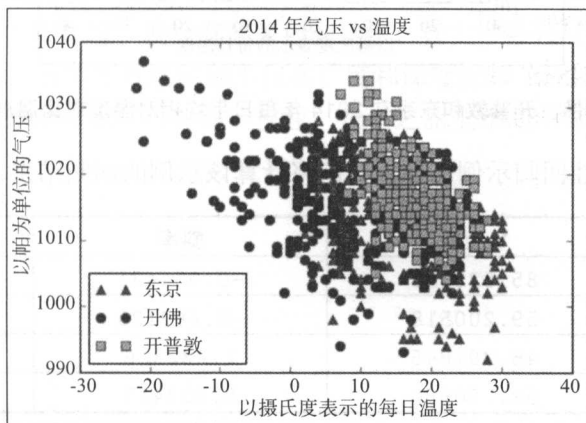


使用简单线性回归的纯Python实现,可以从<https://code.activestate.com/recipes/578914-simple-linear-regression-with-pure-python/>上获得。该实现使用皮尔森积矩相关系数计算回归,以下表格展示了每个城市的计算结果:

城市	Y 坐标	斜率	r
开普敦	3.0738312142044553	0.6096462268894843	0.8380281574300148
丹佛	-5.110855159000105	0.7013438731440127	0.8441417541071602
东京	-5.261920573865529	1.1013535521237958	0.946832088113318
总体	-4.482081471645017	0.9542228667781014	0.880891021556839

通过对线性回归结果的解读(其中应变变量 y 表示露点, 自变量 x 表示温度), 我们的回归模型展示了很强的相关性系数, 整体的简单线性回归模型相关性系数达到了 0.88。

现在, 我们来研究气压作为应变变量同时温度作为自变量时两者之间的关系, 首先将两者绘制如下:



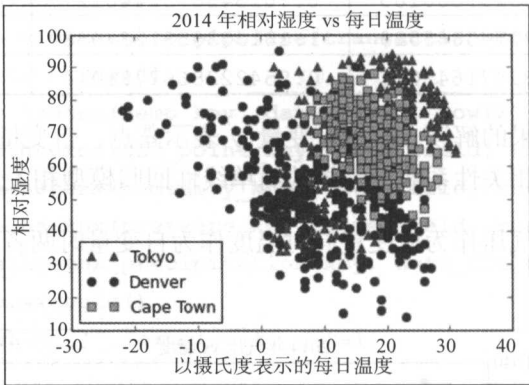
气压和温度的简单线性回归

看起来气压和温度之间没有正向关系, 下列表格展示了每个城市的简单线性回归的计算结果及所有城市组合起来的总体结果。

城市	Y 坐标	斜率	r
开普敦	1029.483421	-0.730263	-0.580355
丹佛	1018.148958	-0.266547	-0.3866715
东京	1019.549379	-0.410279	-0.450956
总体	890.079774	-14.4720573	-0.254273

对平均气压来说，温度与城市有着松散的相关性-0.25，这意味着对于我们的简单线性回归模型来说，温度相对于气压来说不是一个统计学解释变量。

我们把相对于特定温度下饱和水汽的气水混合比率定义为相对湿度。如果把相对湿度作为应变量同时温度作为自变量，那么较高的相对湿度意味着更大机会的降雨这一假设，图形化如下所示。



丹佛、开普敦和东京在 2014 年每日平均相对湿度对比温度

像前两个简单线性回归示例那样，我们将计算该示例的回归结果，如下表所示。

城市	Y 坐标	斜率	r
开普敦	85.525525	-0.954431	-0.453038
丹佛	59.200518	-0.663125	-0.433978
东京	46.291846	1.232326	0.578158
总体	58.190660	0.303417	0.162543

从结果中我们可以看到，简单线性回归模型无法解释温度和相对湿度之间的关系，至少是在 2014 年的开普敦、丹佛和东京是这样的。所有三个城市的总体数据的相关系数也只有 0.162543，其中有很多数据的变化用线性模型是无法解释的。

总结

本章的焦点又重新回到了本书第 1 章中提到的使用 Redis 的原因上。对于许多 ETL 工作来说，Redis 非常适合用于将不同的数据源“胶合”到最终目标上。我们看到了一个加载 DP.LA 数据集的简单示例。该数据集包含了来自伊利诺伊大学的图像集合和其他内容的元

数据。速度上的差异是由于使用了 Redis 的批量加载选项所带来的数量级上的速度提升。你需要直接从输入数据源上创建 Redis 协议 (RESP)，并在最后使用 Netcat 或者 redis-cli 程序中的特殊模式加载这些数据。我们谈到了用来保护 Redis 实例的最低限度安全策略。在本章的最后，我们研究了两个常见的机器学习技术，即朴素贝叶斯和简单线性回归，并展示了如何使用 Redis 将原始数据流转变为企业的信息和知识。

撰写本书对我这位高校图书管理员的技术能力和 Redis 知识来说是一个挑战，在技术越来越受重视的今天，Redis 在解决日常问题时所具备的应用范围、速度和灵活性令人印象深刻，即便对于图书馆这样“低技术含量”的领域来说也一样！从 Salvatore Sanfilippo 早期尝试开发一个网站缓存开始，他真正的才华在于坚持己见、激情昂扬的领导力，以及众所周知的优秀精良工具 Redis 所体现出来的真正的编程技能。

正是由于 Sanfilippo 的远见卓识和积极开发，Redis 才能持续不断地添加新功能并修复 BUG。如果想要保持对 Redis 的精通，那么你需要持续关注并学习，因为 Redis 的代码库非常活跃，会不断演进下去。即便 Redis 高速发展，大多数现存的功能和命令一旦融入到主 Redis 分支并用于生产环境，就没有显著地变动了。在大多数领域中，“大师”是从来不会停止学习的。因此，我期望现在的你不仅在广度和深度上对 Redis 更加了解，还希望你能在精通像 Redis 那样拥有无尽可能的“快速”技术之旅上不断学习和实验！

附录：来源

第 1 章：为何选择 Redis?

1. 来自 antirez weblog 的 *How to take advantage of Redis just adding it to your stack*, 作者 Salvatore Sanfilippo。详情请访问 <http://oldblog.antirez.com/post/take-advantage-of-redis-adding-it-to-your-stack.html>。

第 2 章：高级键管理与数据结构

1. *An introduction to Redis data types and abstractions*, 详情请访问 <http://redis.io/topics/data-types-intro>。
2. 由 Christopher Stover 贡献的 “Big-O Notation”。该词条来自 Eric W. Weisstein 创建的 *Mathworld--A Wolfram Web Resource*。详情请访问 <http://mathworld.wolfram.com/Big-ONotation.html>。
3. 在 Quora 上由 Animesh Dash 回答的 *What are the differences between memcached and redis?* 详情请访问 <https://www.quora.com/What-are-the-differences-between-memcached-and-redis/answer/Animesh-Dash?srid=Kgp>。
4. *Redis Bitmaps -Fast, Easy, Realtime Metrics*。详情请访问 <http://blog.getspool.com/2011/11/29/fast-easy-realtime-metrics-using-redis-bitmaps/>。

第 3 章：内存管理的建议与技巧

1. 由 Salvatore Sanfilippo 发表的 *Using Redis as an LRU*。详情请访问 <http://redis.io>。

io/topics/ lru-cache。

2. Redis 源代码文件 `redis-cli.c`。可以从 <http://download.redis.io/redis-stable/src/redis-cli.c> 上获取。
3. 由 Salvatore Sanfilippo 发表的 *Using hashes to abstract a very memory efficient plain key-value store on top of Redis*。详情请访问 <http://redis.io/topics/memory-optimization>。
4. 由 Stefan Parvu 发表的 *Raspberry Pi and Redis*。详情请访问 <http://kronometrix.blogspot.com/2014/10/raspberry-pi-and-redis.html>。
5. 由 Riccardo Cecolin 发表的 *Redis Android NDK port*。详情请访问 <http://rikiji.it/2012/08/21/Redis-Android-NDK-port.html>。

第 6 章：可伸缩性：Redis 集群和 Sentinel

1. 由 Salvatore Sanfilippo 在 antirez weblog 上发表的 *Redis Presharding*。详情请访问 <http://oldblog.antirez.com/post/redis-presharding.html>。
2. 由 Marius Przydatek 发表的博客 *Redis data sharding-part 2-hash-based keys*。请访问 <http://mariuszprzydatek.com/2013/08/23/redis-data-sharding-part-2-hash-based-keys/>。
3. *Redis Cluster tutorial*。详情请访问 <http://redis.io/topics/cluster-tutorial>。
4. 由 Salvatore Sanfilippo 发表的 *Twemproxy, a Redis proxy from Twitter*。请访问 <http://antirez.com/news/44>。
5. 邮件归档。详情请访问 <https://groups.google.com/forum/#!msg/redis-db/eTtCNAosiiU/h7ifK2K3FA0J>。

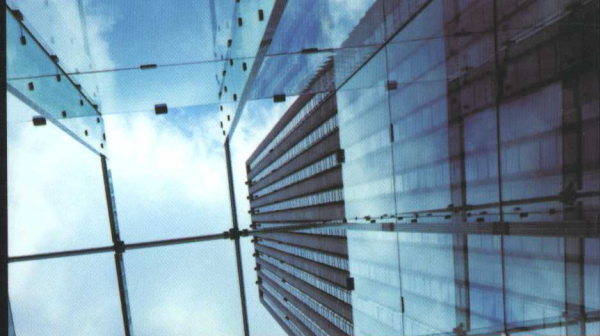
第 7 章：Redis 与互补的 NoSQL 技术

1. *DB-Engines Ranking*。详情请访问 <http://db-engines.com/en/ranking>。
2. 由 Cody Powell 发表的博客 *The Beautiful Marriage of MongoDB and Redis*。详情请访问 <https://dzone.com/articles/beautiful-marriage-mongodb-and>。

3. 由 DJ Walker-Morgan 发表的博客 *Redis, MongoDB & the Power of Incrementency*。详情请访问 <https://www.compose.io/articles/redis-mongodb-and-the-power-of-incremency/>。
4. 由 DJ Walker-Morgan 发表的博客 *Why (and how to) Redis with your MongoDB*。详情请访问 <https://www.compose.io/articles/why-and-how-to-redis-with-your-mongodb/>。
5. 由 Tim Berners-Lee 撰写的 *Linked Data*。详情请访问 <https://www.w3.org/DesignIssues/LinkedData.html>。

第 10 章：信息流的测量与管理

1. *Redis Mass Insertion*。详情请访问 <http://redis.io/topics/mass-insert>。
2. DPLA 块数据可从 <http://dp.la/info/developers/download/> 上进行下载。
3. 由 Kumar、Nitin 于 2012 年 8 月 5 日发表的 *Real time dashboard for redis*。详情请访问 <http://www.nkrode.com/article/real-time-dashboard-for-redis>。
4. 由 Sachin Joglekar 于 2015 年 3 月 7 日发表的 *Efficient computation and storage of basic data statistics using Redis*。详情请访问 <https://codesachin.wordpress.com/2015/07/03/efficient-computation-and-storage-of-basic-data-statistics-using-redis/>。
5. 由 Rick Bennett、Brian F. Lavoie、Edward T. O'Neill 三人于 2003 年联合发表的 *The Concept of a Work in WorldCat: An Application of FRBR*。可从 http://www.oclc.org/content/dam/research/publications/library/2003/lavoie_frbr.pdf 上获取。



深入理解Redis

Redis是当下最流行的开源键值数据结构服务器。它提供了多种功能，可在此之上构建多种平台。

本书定位为实用指南，旨在帮助读者深入理解Redis数据结构，以便充分发挥Redis的优秀功能。读者的Redis之旅始于对Redis需求的讨论，然后讲解了高级键管理方面的内容。接下来，读者将学习设计模式、在DevOps环境中使用Redis的最佳实践，以及Docker容器化范式。在这之后，读者将学习如何使用Redis集群和Redis Sentinel进行扩展，随后将对Redis与其他NoSQL技术（如ElasticSearch和MongoDB）的结合进行说明。最后，读者将了解如何使用Redis为不相同的数据流构建实时数据分析仪表板。

本书的目标读者

如果你是一位有Redis经验的软件开发者，并且想提升Redis的知识和技能，那么本书就是为你准备的。

本书将带领你

- ◎探索Redis 3.2中的新增功能
- ◎选择正确的Redis数据结构解决问题
- ◎理解Redis事件循环并实现自定义C命令
- ◎使用Redis服务器端脚本Lua解决复杂的工作流问题
- ◎配置Redis实例以达到最佳内存管理
- ◎使用Redis集群实现数据的分布式
- ◎使用Redis Sentinel提升Redis解决方案的稳定性
- ◎将Redis用作现存数据库和NoSQL环境的补充方案
- ◎充分利用Redis提供的各种功能，成为一位DevOps专家

